版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

仅供非商业用途或交流学习使用

Broadview®
www.broadview.com.cn

Spring Boot 2 + Thymeleaf 企业应用实战

杨恩雄 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



杨恩雄，从事十多年Java EE企业应用开发，曾任中企动力系统设计师、中企开源项目经理、数码辰星科技公司项目经理，参与多个企业OA、ERP项目的设计与架构工作，在大数据应用、SOA、互联网产品方面有着丰富的开发经验及架构经验，曾著《疯狂工作流讲义》《疯狂Spring Cloud微服务架构实战》等书籍。

扫码关注“疯狂图书”公众号，可获得：

- 本书配套代码
- 本书相关视频
- 作者不定期答疑

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Spring Boot 2 + Thymeleaf 企业应用实战

杨恩雄 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

内 容 简 介

Java 经过多年的发展，市面上出现的开发框架多不胜数，技术发展日新月异。每一次技术的更替都牵动着广大 Java 程序员的心。

在本书的前面部分，讲解目前市面上流行的 Spring Boot 框架，使用该框架我们可以轻松构建出各种项目。在此基础上，介绍 Spring MVC 的使用。学习完这些知识后，读者可以掌握 Spring Boot、Spring MVC 的核心内容。在本书的第二部分，会深入讲解 Thymeleaf 框架，这是一个较为新颖的模板引擎。使用 Thymeleaf，可以完全抛弃 JSP，进入一种更为简便的页面开发模式。第 11 章讲解目前市面上常见的几个 JavaScript 框架，例如 jQuery、Bootstrap 及 Vue.js。学习完这些知识后，读者可以使用这些框架开发大部分的页面功能。第 12 章介绍如何使用 Spring Data 框架操作各个主流数据库，例如 MySQL、Redis 和 MongoDB 等。第 13 章介绍如何使用 Spring Cloud 中的主要框架，学完本章知识后，读者可以开发最基本的微服务项目。本书的最后一章，以一个案例对所学知识进行运用。

本书定位为一本 Java Web 开发的入门级书籍，从表现层到持久层，力求为大家提供一整套 Java Web 开发的技术教程。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Spring Boot 2+Thymeleaf 企业应用实战 / 杨恩雄编著. —北京：电子工业出版社，2018.9
ISBN 978-7-121-34924-9

Ⅰ. ①S… Ⅱ. ①杨… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2018）第 196828 号

策划编辑：张月萍
责任编辑：牛　勇
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：19.75　　字数：530 千字
版　　次：2018 年 9 月第 1 版
印　　次：2018 年 9 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



前　言

Java 经过多年的发展，市面上出现的开发框架多不胜数，技术发展日新月异。程序员群体是一个求知若渴的群体，也是一个容易接受新知识的群体，为了学习新技术，多少人食不知味、夜不能寐。笔者有幸成为一名平凡的程序员，从业十余年，面对各种新技术，时常怀着一颗谦卑的心，不停前进，不断学习。

笔者从 2006 年开始进行 Java Web 开发，使用过大部分主流的 Java 技术，例如 EJB、Servlet、MVC 框架等，在此过程中开发过多个大型项目，经历了这些项目从无到有的过程，深知系统开发之不易。一直以来，期望能使用最简单的方式开发出最完美的系统。近年来随着 Spring Boot 的兴起，Java 的 Web 开发趋向简单，笔者希望能将 Spring Boot 等技术进行推广，从而降低 Java Web 开发的门槛，因此编纂本书。笔者学识浅薄，加之仓促付梓，书中错漏定然难免，望读者指正。

本书能得以出版，得益于多方襄助，对他们的感激之情，难以言表。感谢传道并解惑的恩师，感谢聪颖而好学的读者，感谢善良和亲爱的家人，笔者会谢意永存、铭感不忘。

本书特点

笔者长期工作于企业的 IT 部门，有着丰富的企业应用开发经验，因此本书有以下特点。

1. 涵盖多个技术框架

笔者期望本书能给读者提供一整套的企业应用开发方案，因此本书介绍了目前最为流行的 Spring Boot、Thymeleaf、Spring Data 以及多个 JavaScript 框架。学习完本书知识后，读者掌握的并不是一两个技术框架，而是一整套当前非常流行的开发技术。

2. 内容深入

本书以 Spring Boot 与 Thymeleaf 为核心，其中 Thymeleaf 是目前非常流行的模板框架，本书对该框架进行了系统讲解，内容由浅入深，务求使读者充分掌握这个模板框架。

3. 案例贴近实际，视频带领入门

笔者长期供职于 IT 企业，奋斗于研发一线，因此本书的各个知识点对应的案例更加贴近实际。看得懂、学得会、做得出是笔者一贯的风格，对于初次接触这些知识的读者，本书提供了配套的视频，助力你更快地掌握知识。

4. 注释详细

简单易懂是笔者一直以来坚持的目标，本书中的几乎每行 Java 代码均有注释，读者可以很容易地了解代码的意思，轻松掌握知识。

本书写给谁看

笔者一直以来希望能将复杂的技术简单化，因此将本书定位成一本入门级的 Java Web 开发书籍，只要你有一定的 Java 语言基础，就可以在本书的基础上，学习使用 Spring Boot+Thymeleaf



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

进行 Java Web 应用开发。

衷心感谢

首先非常感谢李刚老师，一直以来，他既是我的老师，也是我的技术后盾，非常幸运人生能有这样一位良师益友。

其次感谢出版社编辑，为本书提出了许多宝贵的意见。

最后感谢我的家人，你们是我前进的动力。

下载本书代码与联系作者

本书的所有代码、软件，均可以到以下网盘获取：http://pan.baidu.com/s/1sl5FWK1。读者也可以扫描以下的二维码，关注“疯狂图书”公众号获取本书资料。公众号内除了提供本书代码外，还提供了相关视频供读者下载。



如在学习过程中有技术相关的问题，可以发邮件到笔者的邮箱：yangenxiong@163.com。

2018 年 7 月 22 日于广州

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

目 录 CONTENTS

第 1 章 概述 1

1.1 Java EE 开发技术 2
1.1.1 Java EE 的三层架构 2
1.1.2 MVC 框架 2
1.1.3 数据访问层框架 3
1.1.4 视图技术 3
1.2 JDK 的下载与安装 3
1.3 安装与配置 Maven 4
1.3.1 关于 Maven 4
1.3.2 下载与安装 Maven 4
1.3.3 配置远程仓库 5
1.4 安装 Eclipse 5
1.4.1 Eclipse 版本 5
1.4.2 在 Eclipse 中配置 Maven 5
1.4.3 获取本书代码 6
1.5 本章小结 6

第 2 章 初试 Spring Boot 7

2.1 Spring Boot 介绍 8
2.1.1 Spring Boot 简介 8
2.1.2 starter 模块 8
2.2 构建第一个 Spring Boot 程序 9
2.2.1 新建 Maven 项目 9
2.2.2 编写启动类 10
2.2.3 编写控制器 11
2.2.4 开发环境的热部署 12
2.3 运行单元测试 12
2.3.1 测试 Web 服务 12
2.3.2 模拟 Web 测试 13
2.3.3 测试业务组件 13
2.3.4 模拟业务组件 14
2.4 发布与调用 REST 服务 15
2.4.1 REST 15
2.4.2 发布 REST 服务 15
2.4.3 使用 RestTemplate 调用服务 16
2.4.4 使用 Feign 调用服务 18
2.5 本章小结 19

第 3 章 Spring Boot 配置 20

3.1 配置文件 21
3.1.1 默认配置文件 21
3.1.2 指定配置文件 22
3.1.3 yml 文件 22
3.1.4 使用 profile 指定配置 23
3.2 Web 配置 24
3.2.1 Servlet 配置 24
3.2.2 Listener 配置 25
3.2.3 Filter 配置 26
3.2.4 构建可部署的 war 包 26
3.2.5 JSP 配置 28
3.3 其他配置 30
3.3.1 服务器常用配置 30
3.3.2 响应压缩 30
3.3.3 SSL 配置 32
3.3.4 使用其他服务器 32
3.3.5 服务器访问日志 33
3.3.6 banner 配置 34
3.4 自定义配置 35
3.4.1 @Value 注解 35
3.4.2 定义属性映射对象 36
3.4.3 使用配置 37
3.4.4 属性自动识别 38
3.5 本章小结 39

第 4 章 Spring Boot 的注解 40

4.1 Spring 的常用注解 41
4.1.1 bean 定义 41
4.1.2 依赖注入 42
4.1.3 使用 Primary 注解 44
4.1.4 Scope 注解 45



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

4.1.5 方法注入 46
4.1.6 AOP 注解 47
4.1.7 ComponentScan 注解 49
4.2 高级 Spring 注解 50
4.2.1 限定注解 50
4.2.2 自定义限定注解 51
4.2.3 自定义 bean 的生命周期 52
4.3 Spring MVC 的常用注解 54
4.3.1 Controller 注解 54
4.3.2 RequestMapping 注解 55
4.3.3 PathVariable 注解 56
4.3.4 MatrixVariable 注解 56
4.3.5 RequestParam 注解 57
4.3.6 文件上传 58
4.3.7 RestController 注解 58
4.4 Spring Boot 的条件注解 59
4.4.1 类条件注解 59
4.4.2 项目依赖问题 60
4.4.3 其他条件注解 61
4.4.4 自定义条件注解 61
4.4.5 Spring Boot 的自动配置 62
4.5 本章小结 63

第 5 章 初试 Thymeleaf 64

5.1 Thymeleaf 65
5.1.1 什么是 Thymeleaf 65
5.1.2 支持的模板类型 65
5.1.3 HTML 模板 65
5.1.4 XML 模板 66
5.2 初试 Thymeleaf API 66
5.2.1 处理 HTML 66
5.2.2 更换模板解析器 67
5.2.3 处理资源文件 67
5.2.4 变量处理 68
5.2.5 遍历集合 69
5.2.6 设置前缀与后缀 69
5.3 在 Servlet 应用中整合 Thymeleaf 70
5.3.1 建立项目 70
5.3.2 配置模板引擎 71
5.3.3 在 Servlet 中使用 Thymeleaf 72
5.3.4 编写模板文件 73
5.4 在 Struts2 中整合 Thymeleaf 74
5.4.1 建立与配置 Struts2 项目 74
5.4.2 配置 Struts Action 75
5.4.3 整合 Themeleaf 76
5.5 在 Spring Boot 中整合 Thymeleaf 77
5.5.1 建立与配置项目 77
5.5.2 编写控制器与模板 78
5.5.3 静态资源 79
5.6 本章小结 79

第 6 章 Thymeleaf 对象的使用 80

6.1 文本处理 81
6.1.1 外部文本输出 81
6.1.2 文本国际化 82
6.1.3 处理转义文本 83
6.1.4 读取变量 83
6.1.5 在文本信息中使用变量 84
6.1.6 其他读取变量的方式 84
6.1.7 链接表达式 85
6.2 基本对象 86
6.2.1 基本对象概述 86
6.2.2 #ctx 对象 87
6.2.3 Web 对象 87
6.2.4 内置变量 88
6.2.5 使用基本对象的注意事项 88
6.3 数字对象 89
6.3.1 整数格式化 89
6.3.2 小数格式化 90
6.3.3 货币格式化 91
6.3.4 百分比格式化 92
6.3.5 使用 sequence 方法 92
6.4 字符串对象 92
6.4.1 toString 与 length 方法 93
6.4.2 非空判断与默认值处理 93
6.4.3 包含判断 94
6.4.4 截取与替换 94
6.4.5 追加与拼接 95

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

6.4.6 分割与连接 96
6.4.7 大小写转换 96
6.4.8 字符串编码与解码 97
6.4.9 其他处理 98
6.5 日期对象 98
6.5.1 格式化日期 98
6.5.2 获取日期字段 99
6.5.3 创建日期 100
6.6 数组与集合对象 101
6.6.1 数组 101
6.6.2 List 与 Set 101
6.6.3 Map 101
6.7 其他内置对象 102
6.7.1 messages 对象 102
6.7.2 uris 对象 102
6.7.3 aggregates 对象 103
6.8 本章小结 103

第 7 章 Thymeleaf 常用语法 104

7.1 表达式语法之运算符 105
7.1.1 表达式常量 105
7.1.2 字符串拼接 106
7.1.3 算术运算符 106
7.1.4 关系运算符 107
7.1.5 条件运算符 108
7.1.6 无操作符 108
7.2 表达式语法进阶 108
7.2.1 数据转换 108
7.2.2 自定义数据转换类 109
7.2.3 Spring Boot 数据转换 110
7.2.4 表达式预处理 112
7.2.5 表达式调用工具方法 113
7.3 属性设置 114
7.3.1 使用 th:attr 114
7.3.2 设置 HTML 节点属性 114
7.3.3 属性值拼接 115
7.3.4 HTML5 属性支持 115
7.4 条件判断与数据迭代 116
7.4.1 条件判断 116
7.4.2 switch case 语法 117
7.4.3 使用 th:each 属性 117
7.4.4 常见迭代类型 118
7.4.5 迭代状态对象 118
7.4.6 数据延迟加载 120
7.4.7 在 Spring Boot 中使用延迟加载 121
7.4.8 星号表达式 121
7.5 本章小结 122

第 8 章 深入 Thymeleaf 模板 123

8.1 模板片断 124
8.1.1 定义与引用模板片断 124
8.1.2 片断引用语法 125
8.1.3 选择器的基础语法 126
8.1.4 选择器的其他语法 128
8.1.5 insert、replace 与 include 的区别 128
8.1.6 含有变量的片断引用 129
8.1.7 片断块引用 130
8.1.8 无操作符的使用 131
8.1.9 删除模板 132
8.1.10 基本表达式总结 134
8.2 模板与逻辑分离 134
8.2.1 逻辑分离例子 134
8.2.2 逻辑分离配置 136
8.2.3 关联节点的选择 137
8.3 模板注释 137
8.3.1 解析层注释 138
8.3.2 原型注释 139
8.4 内联语法 139
8.4.1 使用内联语法显示文本 139
8.4.2 在 JavaScript 中使用内联语法 140
8.4.3 在 CSS 中使用内联语法 140
8.4.4 内联语法注释 141
8.4.5 内联语法序列化 142
8.5 模板缓存 143
8.5.1 开启与配置缓存 143
8.5.2 缓存有效时间 145
8.6 本章小结 146



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 9 章 Thymeleaf 原理与扩展 147
9.1 概念与接口介绍 148
9.1.1 方言 148
9.1.2 处理器 149
9.1.3 模板处理者 150
9.1.4 模板解析事件 150
9.1.5 节点与标签 151
9.2 深入 Thymeleaf 模板处理 151
9.2.1 策略模式 151
9.2.2 责任链模式 153
9.2.3 合并使用策略模式与责任链模式 ... 156
9.2.4 模板处理过程 159
9.2.5 模板事件 161
9.3 处理器 161
9.3.1 预处理器 161
9.3.2 后处理器 163
9.3.3 一般处理器 164
9.3.4 处理器优先级 165
9.3.5 标签处理器 167
9.3.6 模型处理器 168
9.3.7 边界处理器 170
9.3.8 其他处理器 170
9.4 Thymeleaf 扩展 171
9.4.1 自定义标签 171
9.4.2 自定义模板属性 173
9.4.3 自定义内置对象 175
9.4.4 自定义执行属性 176
9.5 本章小结 177

第 10 章 Spring Boot 与 Thymeleaf 整合 178
10.1 在 Spring Boot 中配置 Thymeleaf 179
10.1.1 模板引擎配置 179
10.1.2 Spring MVC 视图解析器 181
10.1.3 Thymeleaf 视图解析器 184
10.1.4 数据转换配置 186
10.1.5 模板国际化 186
10.2 表单处理 188
10.2.1 表单提交 188
10.2.2 使用 th:field 属性 189
10.2.3 HTML 节点的 id 生成 190
10.2.4 实现 label 节点 192
10.2.5 在模板中处理 checkbox 与 radio ... 193
10.2.6 表单提交处理 checkbox 与 radio ... 195
10.3 表单验证 195
10.3.1 表单验证 195
10.3.2 修改信息文件 197
10.3.3 输出错误信息 197
10.4 片断的使用 198
10.4.1 引用片断 198
10.4.2 使用 bean 选择片断 199
10.4.3 在控制器中引用片断 201
10.5 自定义标签 201
10.5.1 方言与处理器 201
10.5.2 配置方言 203
10.6 本章小结 204

第 11 章 使用 JavaScript 库 205
11.1 使用 jQuery 206
11.1.1 概述 206
11.1.2 选择器 206
11.1.3 事件 207
11.1.4 数据列表 208
11.1.5 AJAX 调用 GET 方法 209
11.1.6 AJAX 调用 POST 方法 210
11.1.7 表单验证 211
11.2 Bootstrap 213
11.2.1 概述 213
11.2.2 数据列表 214
11.2.3 分页 215
11.2.4 表单 218
11.2.5 警告框的使用 219
11.3 Vue.js 221
11.3.1 概述 221
11.3.2 组件 222
11.3.3 指令 224
11.3.4 循环指令 225
11.3.5 表单验证 226
11.4 本章小结 227

第 12 章　数据库实战.....228
12.1　概述.....229
12.1.1　关于 Spring Data.....229
12.1.2　Spring Data 的功能.....229
12.1.3　Spring Data 模块.....230
12.2　Spring Data 与 JPA.....230
12.2.1　构建项目.....230
12.2.2　数据访问层与业务层.....231
12.2.3　自定义数据存储逻辑.....233
12.2.4　方法名查询.....233
12.2.5　使用@Query 注解.....234
12.3　Spring Data 与 MongoDB.....235
12.3.1　安装 MongoDB.....235
12.3.2　配置权限.....236
12.3.3　MongoDB 的相关概念.....236
12.3.4　构建项目.....236
12.3.5　数据访问层与业务层.....237
12.3.6　自定义数据存储逻辑.....238
12.3.7　方法名查询.....239
12.3.8　使用@Query 注解.....240
12.4　Spring Data 与 Redis.....241
12.4.1　Redis 的安装与配置.....241
12.4.2　Redis 的数据类型.....242
12.4.3　使用 Jedis.....242
12.4.4　构建 Spring Data 项目.....243
12.4.5　数据访问层与业务层.....244
12.4.6　自定义数据存储逻辑.....246
12.4.7　方法名查询.....247
12.5　本章小结.....247

第 13 章　开发微服务.....248
13.1　微服务与 Spring Cloud.....249
13.1.1　微服务简介.....249
13.1.2　Netflix 与 Spring Cloud.....249
13.1.3　Spring Cloud 模块介绍.....250
13.2　Eureka 服务治理.....250
13.2.1　Eureka 架构.....250
13.2.2　构建服务器.....251
13.2.3　服务提供者.....253
13.2.4　服务调用者.....255
13.2.5　程序结构.....257
13.3　Ribbon 负载均衡.....257
13.3.1　Ribbon 介绍.....258
13.3.2　第一个 Ribbon 程序.....258
13.3.3　在 Spring Cloud 中测试使用 Ribbon.....259
13.4　客户端 Feign.....260
13.4.1　Feign 介绍.....260
13.4.2　第一个 Feign 程序.....260
13.4.3　使用 Feign 调用微服务.....262
13.5　容错框架 Hystrix.....263
13.5.1　Hystrix 介绍.....263
13.5.2　第一个 Hystrix 程序.....265
13.5.3　在 Spring Cloud 中整合 Hystrix.....266
13.6　本章小结.....268

第 14 章　实战案例.....269
14.1　系统功能及界面.....270
14.1.1　登录与首页.....270
14.1.2　图书管理模块.....271
14.1.3　入库模块.....272
14.1.4　销售模块.....273
14.1.5　导航组件与分页组件.....274
14.1.6　运行本章案例.....275
14.2　数据库设计.....275
14.2.1　用户表.....275
14.2.2　图书表与库存表.....275
14.2.3　入库单数据表.....276
14.2.4　销售单数据表.....276
14.3　搭建项目框架.....277
14.3.1　项目依赖与项目结构.....277
14.3.2　项目配置.....278
14.3.3　实体设计.....279
14.3.4　用户登录.....281
14.3.5　片断定义.....282
14.4　图书管理.....283
14.4.1　数据列表.....283
14.4.2　通用的分页组件.....285

14.4.3 图片上传 286
14.4.4 新建图书 288
14.4.5 查看与修改图书 289
14.4.6 删除图书 290
14.5 销售单模块 291
14.5.1 列表与分页功能 291
14.5.2 选择图书 293
14.5.3 删除已选图书与修改数量 295
14.5.4 新建销售单 297
14.5.5 操作库存 299
14.6 首页数据查询与排序 300
14.6.1 库存查询 300
14.6.2 图书查询与轮播 301
14.6.3 数据排序 303
14.7 本章小结 304

CHAPTER

1

第1章 概 述

本章要点

- Java EE 开发技术介绍
- Eclipse
- 构建工具 Maven

Java 经过多年的发展，市面上的开发框架多不胜数，技术发展日新月异。本书将会以 Spring Boot、Thymeleaf 与 Spring Data 整合开发为主线，讲述一套全新的 Java EE 开发技术。

本章作为第一章，会首先对 Java EE 的开发技术做一个简单的概述。为了避免读者在学习知识的过程中出现障碍，本章会搭建开发环境，介绍本书所使用的开发工具，如果读者已经掌握了这些工具，则可跳过本章，直接学习后面的知识。

1.1 Java EE 开发技术

首先来了解目前 Java EE 领域一些常用的开发框架。

1.1.1 Java EE 的三层架构

经典的 Java EE 应用是三层架构，分别为表现层、业务逻辑层和数据访问层。这种应用的划分方式，可以让项目的参与人员发挥各自的专长，更加专注于特定的开发工作。经典的 Java EE 三层架构如图 1-1 所示。

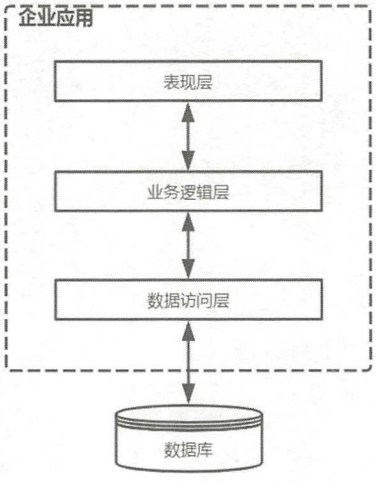


图 1-1　Java EE 应用三层架构

在过去的十年间，这个经典的架构被应用于各种企业级项目中，深得开发者的青睐，精通三层开发技术几乎成为 Java 程序员的标配。这么多年来，基于这个架构，涌现出了不少优秀的开发框架，下面将介绍目前市面上较为流行的开发框架。

1.1.2 MVC 框架

说起 Servlet 框架，不得不说最经典的 Struts。它是一个基于 Servlet/JSP 的 MVC 框架，采用了经典的 MVC 模式。它的诞生，大大减轻了 Web 开发的工作量，一度成为 Java Web 领域中最流行的框架。但是由于 Struts 的部分缺陷（例如过于依赖 Servlet），造成很多开发人员转去使用其他的 MVC 框架（例如 WebWork、Tapestry）。Struts 在发展的过程中，结合了 WebWork 框架，从而推出了全新的 Struts2。实际上，Struts2 已经抛弃了 Struts1 的核心，完全采用 WebWork 的核心设计。目前 Struts2 仍然是主流的 MVC 框架之一。

除了 Struts 之外，Java EE 领域另一个佼佼者 Spring，也推出了自己的 MVC 框架：Spring Web MVC。这是一个基于 Servlet 构建的 Web 框架，刚推出时虽然并没有得到广泛的应用，但

随着它不断的发展，现在俨然成为 MVC 框架的首选。Spring 在刚开始，只负责提供 IoC 容器，用于整合、管理企业应用中的各个组件。之前可能有朋友将 Spring 划分到业务逻辑层中，认为它是一个业务逻辑层的框架，但随着 Spring 涉猎的领域越来越广，其不仅推出了 Spring MVC，还推出了数据访问层框架 Spring Data。Spring Data 的作用更像是一个数据访问层的适配器。这样来看，Spring 大有“一统江湖”之势。本书后面的章节，也会涉及 Spring Data 的相关知识。

以上简单介绍了 Struts 和 Spring MVC，除了它们之外，还有诸如 Tapestry、JSF 等 MVC 框架，在此就不展开讨论了。由于本书以 Spring Boot 为核心，因此本书会使用 Spring MVC 框架。

1.1.3 数据访问层框架

数据访问层主要对数据进行增、删、改、查。最初人们使用 JDBC 的 API 进行数据库操作，后来，著名程序员 Gavin King 创建了 Hibernate 框架，该框架对 JDBC 进行了封装，让 Java 对象与数据库表建立映射关系，操作对象等于操作数据，这使得 Hibernate 迅速成为数据访问层的热门框架。除了 Hibernate 之外，另一个开源框架 MyBatis 也是业界的宠儿，一旦涉及关系型数据库的项目，大家首先想到的就是这两个开源框架。随着 NoSQL 数据库的兴起，目前在数据访问层，我们拥有更多的技术选项。Spring 针对数据访问层，提供了 Spring Data 框架，这等于在各个数据访问框架之上，增加了一个适配层，其目的是使用同样的模型、代码来实现不同数据库操作。本书将主要讲解 Spring Data 框架。

1.1.4 视图技术

在传统的 Java Web 开发中，多使用 JSP 视图，但也有人使用 Freemarker、Velocity 等模板技术。Spring MVC 支持多种模板技术，例如 Freemarker、Thymeleaf、Groovy Markup 等。Freemarker 作为一个老牌的模板引擎，目前市面上已有非常多的文档可以参考。相对于 Freemarker 来说，Thymeleaf 较为新颖，本书将以 Thymeleaf 为核心，深入讲解 Thymeleaf 技术。

本节主要向大家简单介绍了 Java EE 三层架构及相应的主流技术，再次申明，本书将会以 Spring Boot、Thymeleaf、Spring Data 整合为主线，讲解全新的 Java EE 开发技术。下面开始搭建本书的开发环境，如果读者已经掌握相关的工具，可直接跳过下面的内容。

1.2 JDK 的下载与安装

在本书中，使用的操作系统为 64 位的 Windows 7，使用 64 位的 JDK10。读者可以自行到 Oracle 官方网站下载 JDK，也可以访问 http://pan.baidu.com/s/1sl5FWK1 下的 soft 目录。JDK 安装文件名称为 jdk-10.0.1_windows-x64_bin.exe。安装 JDK 后，添加 Windows 环境变量，如图 1-2 所示。

修改 Path 变量，在变量值后追加“%JAVA_HOME%\bin”，如图 1-3 所示。



图 1-2 JAVA_HOME

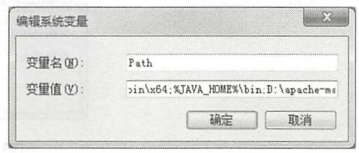


图 1-3 配置 Path

配置完成后，在命令行窗口输入 java –version，输出如图 1-4 所示。

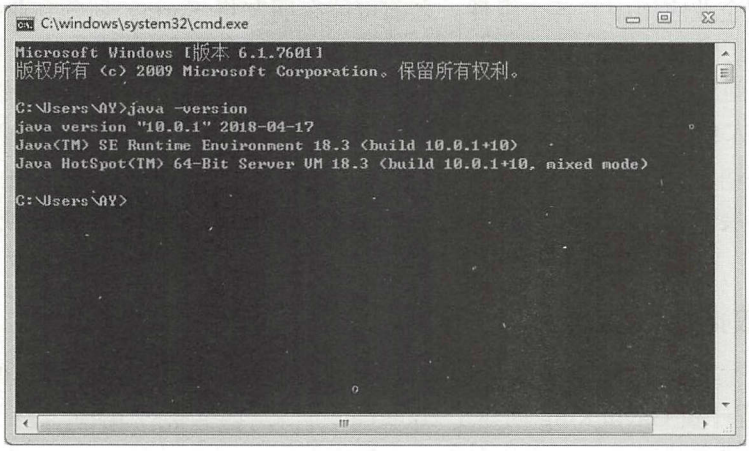


图 1-4　查看 Java 版本

1.3　安装与配置 Maven

1.3.1　关于 Maven

Maven 是 Apache 下的一个开源项目，用于项目的构建。使用 Maven 可以对项目的依赖包进行管理，它也支持构建脚本的继承。对于一些模块（子项目）较多的项目来说，Maven 是更好的选择，子项目可以继承父项目的构建脚本，从而减少了构建脚本的冗余。

除此之外，Maven 本身的插件机制也使其更加强大和灵活，使用者可以配置各种 Maven 插件来完成自己的事情。如果感觉官方或者第三方提供的 Maven 插件不够用，还可以编写符合自己要求的 Maven 插件。Maven 为使用者提供了一个统一的依赖仓库，各种开源项目的发布包都可以在上面找到。在一间公司或者一个项目组内部，甚至可以搭建私有的 Maven 仓库，将自己项目的包放到私有仓库中，供其他项目组或者开发者使用。

在 Maven 的众多特性中，最为重要的是它对依赖包的管理，Maven 将项目所使用的依赖包的信息放到 pom.xml 的 dependencies 节点。例如我们要使用 spring-core 模块的 jar 包，只需在 pom.xml 中配置该模块的依赖信息，Maven 会自动将 spring-beans 等模块引入我们项目的环境变量中。

1.3.2　下载与安装 Maven

本书所使用的 Maven 版本为 3.5，可以到 Maven 官方网站下载：http://maven.apache.org/，也可以到本书的 soft 目录下，找到文件名为 apache-maven-3.5.0-bin.zip 的包。下载并解压后得到 apache-maven-3.5.0 目录，将主目录下的 bin 目录加入到系统的环境变量中，如图 1-5 所示。

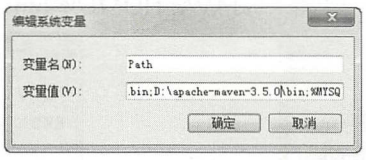


图 1-5　修改环境变量

配置完成后，打开 cmd 命令行，输入“mvn –v”，稍后可以看到输出的 Maven 版本信息。

Maven 下载的依赖包会被存放在本地仓库中，默认路径为 C:\Users\用户名\.m2\repository。

1.3.3 配置远程仓库

如果不进行仓库配置，在默认情况下，会到 Apache 官方的仓库下载依赖包。由于 Apache 官方的仓库位于国外，所以下载速度较慢，这会降低开发效率，笔者建议使用国内的 Maven 仓库或者搭建自己的仓库。本书的重点不是 Maven，因此直接使用了由阿里云提供的 Maven 仓库。修改 apache-maven-3.5.0/conf 目录下的 setting.xml，在 mirror 节点下加入以下配置：

```
<mirror>
    <id>alimaven</id>
    <name>aliyun maven</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
    <mirrorOf>central</mirrorOf>
</mirror>
```

配置完后，以后在使用过程中，Maven 会先到阿里云的仓库下载依赖包。

1.4 安装 Eclipse

1.4.1 Eclipse 版本

本书使用的 Eclipse 版本为 Oxygen（4.7），大家可以从以下的地址得到该版本的 Eclipse：http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygen3a，也可以在上一节提到的 soft 目录下找到该版本的 Eclipse，文件名为 eclipse-java-oxygen-3a-win32-x86_64.zip。

1.4.2 在 Eclipse 中配置 Maven

Luna 版本的 Eclipse 自带了 Maven 插件，默认使用的是 Maven 3.2，由于我们前面安装的是 Maven 3.5 版本，因此需要在 Eclipse 中指定 Maven 版本以及配置文件。按照如图 1-6 所示指定 Maven 的版本，按照如图 1-7 所示指定配置文件。

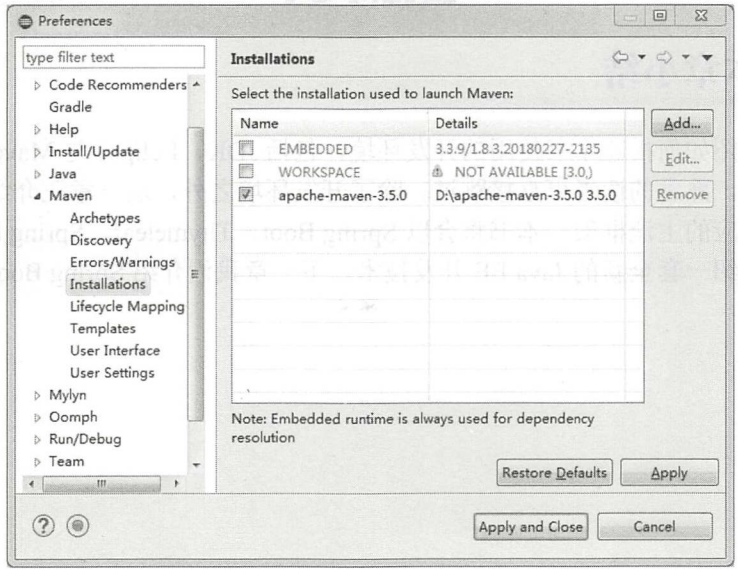


图 1-6 在 Eclipse 中指定 Maven 的版本

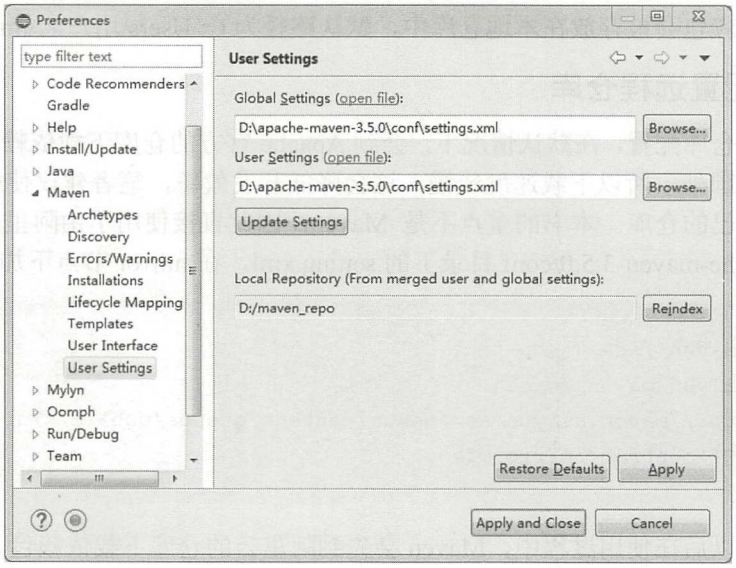


图 1-7 指定 Maven 的配置文件

注意：本书的案例，如无特别说明均以 Maven 项目的形式导入。

1.4.3 获取本书代码

本书的所有代码、软件，均可以到以下网盘获取：http://pan.baidu.com/s/1sl5FWK1。读者也可以扫描以下的二维码，关注“疯狂图书”公众号获取，公众号内除了包含本书代码外，还有相关的配套视频供读者下载。

1.5 本章小结

本章主要介绍如何搭建本书使用的开发环境，包括 JDK、Eclipse 与 Maven 的安装。环境的搭建较为简单，熟悉的读者可直接跳过。除了开发环境之外，这一章还介绍了传统 Java EE 的三层架构及对应的主流框架。本书将会以 Spring Boot、Thymeleaf、Spring Data 整合开发为主线，向大家介绍一套全新的 Java EE 开发技术。下一章我们介绍 Spring Boot。

CHAPTER

2

第2章 初试 Spring Boot

本章要点

- Spring Boot 介绍
- 构建第一个 Spring Boot 程序
- 运行单元测试
- 发布与调用 REST 服务

仅供非商业用途或交流学习使用

Spring Boot 主要提供快速构建项目的功能。我们在本章会使用 Spring Boot 构建第一个 Web 程序，同时介绍 Spring Boot 最简单的功能，例如运行单元测试，发布与调用 REST 服务等。

2.1 Spring Boot 介绍

我们先来了解一下 Spring Boot 项目。

2.1.1 Spring Boot 简介

开发一个全新的项目，需要先搭建开发环境，例如确定要使用的技术框架及版本，还要考虑各个框架之间的版本兼容问题。完成这些烦琐的工作后，还要对新项目进行配置，测试其能否正常运行，最后才将搭建好的环境提交给项目组的其他成员使用。经常会出现的情形是，项目表面上已经成功运行，但部分项目组成员仍然无法运行项目。对于每一个项目，在初期都会浪费大量的时间做这些固定的事情。

受 Ruby On Rails、Node.js 等技术的影响，Java EE 领域需要一种更为简便的开发方式，来取代这些烦琐的项目搭建工作。在此背景下，Spring 推出了 Spring Boot 项目，该项目可以让使用者更快速地搭建项目，从而使用者可以更专注于业务系统的开发。系统配置、基础代码、项目依赖的 jar 包，甚至开发时所用到的应用服务器等，Spring Boot 都可以帮我们准备好。只要在建立项目时，使用构建工具加入相应的 Spring Boot 依赖包，项目即可运行，使用者无须关心版本兼容等问题。

Spring Boot 支持 Maven 和 Gradle 这两款构建工具。Gradle 使用 Groovy 语言编写构建脚本，与 Maven、Ant 等构建工具有良好的兼容性。鉴于笔者使用 Maven 较多，因此本书使用 Maven 作为项目构建工具。本书成书时，Spring Boot 最新的正式版本为 2.0.1，其要求 Maven 版本为 3.2 或以上。

2.1.2 starter 模块

Spring Boot 提供了许多 starter 模块，starter 为我们提供了“一站式”服务，在项目中另加入对应框架的 starter 的依赖，可以免去我们到处寻找依赖包的烦恼。只需要加入一个依赖，项目就可以运行，这就是 starter 的作用。例如，如果你想让你的项目拥有访问关系型数据库的能力，则只需要在你的项目中加入 spring-boot-starter-data-jpa 依赖就可以实现：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <version>2.0.1.RELEASE</version>
</dependency>
```

Spring Boot 官方提供的 starter 模块，命名规则为“spring-boot-starter-*”，其中“*”代表对应的应用类型，这些 starter 的名称，可以帮助我们快速地找到相应的依赖。如果想构建自己的 starter 模块，官方建议命名规则为“project-spring-boot-starter”。本书的很多章节都会使用 starter 模块。

下面介绍一些比较常用的 starter 模块。

- spring-boot-starter-web：用于构建 Web 应用的 starter 模块，加入该依赖后，会包含

仅供非商业用途或交流学习使用

Spring MVC 框架，默认内嵌 Tomcat 容器。

- spring-boot-starter-jpa：用于构建 Spring Data JPA 应用，使用 Hibernate 作为 ORM 框架。
- spring-boot-starter-mongodb：用于构建 Spring Data MongoDB 应用，底层使用 MongoDB 驱动操作 MongoDB 数据库。
- spring-boot-starter-redis：用于构建 Spring Data Redis 应用，使用 Jedis 框架操作 Redis 数据库。
- spring-boot-starter-thymeleaf：构建一个使用 Thymeleaf 作为视图的 Web 应用。
- spring-boot-starter-test：顾名思义，这个 starter 模块主要用于进行单元测试。

2.2 构建第一个 Spring Boot 程序

这一节，我们使用 Spring Boot 构建一个最简单的 Web 应用。

2.2.1 新建 Maven 项目

Spring Boot 使用 3.2 以上版本的 Maven，这里我们使用的版本为 3.5。在 Eclipse 的菜单中选择“File→New→Other”命令，选中“Maven”下的“Maven Project”，单击“Next”按钮，在“New Maven Project”页面中，选择“Create a simple project”项，创建一个简单的 Maven 项目。在弹出的创建项目页面中，输入相应的项目信息，如图 2-1 所示。

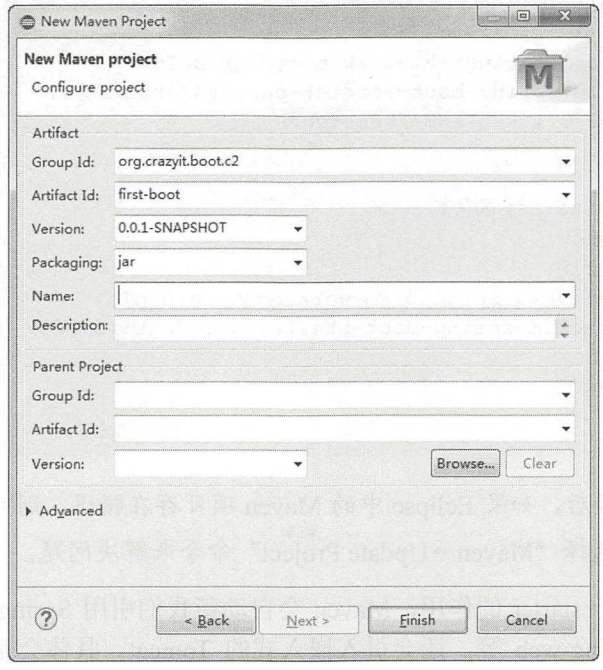


图 2-1 新建 Maven 项目

新建的 Maven 项目，其结构如图 2-2 所示。

仅供非商业用途或交流学习使用

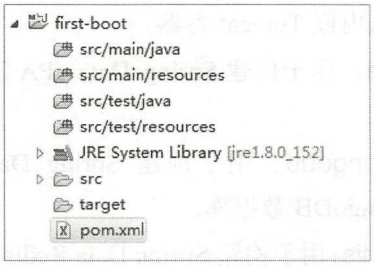


图 2-2 Maven 项目结构

src/main/java 用于存放主应用程序的类，src/main/resources 用于存放主应用程序的资源文件，src/test 用于存放测试相关的 Java 类和资源，pom.xml 则是 Maven 的构建脚本。

一般情况下，Maven 脚本文件需要继承“spring-boot-starter-parent”项目，并在脚本中根据需要声明一个或多个 starter。修改项目的 pom.xml 文件，如代码清单 2-1 所示。

代码清单 2-1：codes\02\first-boot\pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.crazyit.boot.c2</groupId>
    <artifactId>first-boot</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <!-- 继承 spring boot 的 starter -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.1.RELEASE</version>
    </parent>

    <!-- 添加 web starter 的依赖 -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

</project>
```

注意：在加入依赖后，如果 Eclipse 中的 Maven 项目存在错误，则可以选中项目，鼠标右击，在弹出的菜单中选择“Maven→Update Project”命令来解决问题。

加入依赖后，由于 starter 的作用，Maven 会自动帮我们引用 Spring 框架的各个模块，例如 spring-context、spring-web 等，还会引入嵌入式的 Tomcat。具体会帮我们的项目加入哪些依赖包，我们在 Eclipse 下面看一下，有个大概印象即可。

2.2.2 编写启动类

编写一个简单的启动类，就可以直接启动 Web 服务，启动类如代码清单 2-2 所示。

仅供非商业用途或交流学习使用

代码清单 2-2：codes\02\first-boot\src\main\java\org\crazyit\boot\c2\FirstApp.java

```
@SpringBootApplication
public class FirstApp {

    public static void main(String[] args) {
        SpringApplication.run(FirstApp.class, args);
    }

}
```

FirstApp 类使用了@SpringBootApplication 注解，该注解声明该类是一个 Spring Boot 应用，该注解具有@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan 等注解的功能，这些注解会在后面章节讲解。直接运行 MyApp 的 main 方法，看到以下输出信息后，证明启动成功：

```
    2017-10-30 11:40:49.968  INFO 5916 --- [          main]
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
    2017-10-30 11:40:50.199  INFO 5916 --- [          main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
    2017-10-30 11:40:50.208  INFO 5916 --- [          main]
org.crazyit.boot.c2.FirstApp             : Started FirstApp in 6.226 seconds (JVM running
for 6.888)
```

根据输出信息可知，启动的 Tomcat 端口为 8080。打开浏览器访问 http://localhost:8080，可以看到错误页面，表示应用已经成功启动。

注意：在默认情况下，启动端口为 8080，如果需要修改 Tomcat 端口，则可以在 src/main/resources 目录下面，新建 application.properties 文件，通过设置 server.port 属性来配置 Tomcat 端口。具体配置方法在本书后面章节讲解。

2.2.3 编写控制器

前面我们加入了 spring-boot-starter-web 模块，默认集成了 Spring MVC，因此只需要编写一个 Controller，即可实现一个最简单的 HelloWorld 程序。代码清单 2-3 为该控制器的代码。

代码清单 2-3：codes\02\first-boot\src\main\java\org\crazyit\boot\c2\MyController.java

```
@Controller
public class MyController {

    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello World";
    }
}
```

在代码清单 2-3 中，使用了@Controller 注解来修饰 MyController，由于启动类中使用了@SpringBootApplication 注解，该注解具有@ComponentScan 的功能，因此@Controller 会被扫描并注册。在 hello 方法中使用了@GetMapping 与@ResponseBody 注解，以声明 hello 方法的访问地址及返回内容。重新运行启动类，打开浏览器并访问地址 http://localhost:8080/hello，则可以看到控制器的返回内容。

至此，我们的第一个 Spring Boot Web 程序已经完成了，整个过程非常简单。大家可以看

仅供非商业用途或交流学习使用

到，使用 Spring Boot 后，使我们节省了很多搭建项目框架的时间，Spring Boot 的 starter 提供了这种“一站式”的服务，帮助我们开发 Web 应用。

2.2.4 开发环境的热部署

每次修改 Java 后，都需要重新运行 main 方法才能生效，这样会降低开发效率。我们可以使用 Spring Boot 提供的开发工具来实现热部署，为项目加上以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

修改 Java 文件后，容器会重新加载本项目的 Java 类。

2.3 运行单元测试

单元测试对于程序来说非常重要，它不仅能增强程序的健壮性，而且也为程序的重构提供了依据，目前很多开源项目的测试覆盖率高达 90%以上，可见大家对单元测试的重视。根据前面章节可知，Spring Boot 运行 Web 应用，只需要执行 main 方法即可，那么如何测试这个 Web 程序？如何测试 Spring Boot 中的组件呢？这一节，将简单介绍 Spring Boot 的单元测试。

2.3.1 测试 Web 服务

Spring Boot 提供了@SpringBootTest 注解，可以让我们在单元测试中测试 Spring Boot 的程序。先为我们的项目加入“spring-boot-starter-test”依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

新建测试类，用于测试第一个 Spring Boot 程序的“/hello”服务，测试类请见代码清单 2-4。

代码清单 2-4：codes\02\first-boot\src\test\java\org\crazyit\boot\c2\RandomPortTest.java

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testHello() {
        // 测试 hello 方法
        String result = restTemplate.getForObject("/hello", String.class);
        assertEquals("Hello World", result);
    }
}
```

在代码清单 2-4 中，为测试类加入了@RunWith、@SpringBootTest 注解，其中为

仅供非商业用途或交流学习使用

SpringBootTest 配置了 webEnvironment 属性，表示在运行测试时，会为 Web 容器随机分配端口。在测试方法中，使用@Test 注解修饰，使用 TestRestTemplate 调用“/hello”服务。

这个 TestRestTemplate 对象，实际上是对 RestTemplate 进行了封装，可以让我们在测试环境更方便使用 RestTemplate 的功能，例如代码清单 2-4，我们不需要知道 Web 容器的端口是多少，就可以直接进行测试。

在代码清单 2-4 中配置了随机端口，如果想使用固定的端口，则可以将 webEnvironment 配置为 WebEnvironment.DEFINED_PORT。使用该属性，会读取项目配置文件（例如 application.properties）中的端口（server.port 属性）来启动 Web 容器，如果没有配置，则使用默认端口 8080。

2.3.2 模拟 Web 测试

在设置@SpringBootTest 的 webEnvironment 属性时，不管设置为 RANDOM_PORT 还是设置为 DEFINED_PORT，在运行单元测试时，都会启动一个真实的 Web 容器。如果不想启动真实的 Web 容器，则可以将 webEnvironment 属性设置为 WebEnvironment.MOCK，来启动一个模拟的 Web 容器，如代码清单 2-5 所示。

代码清单 2-5：codes\02\first-boot\src\test\java\org\crazyit\boot\c2\MockEnvTest.java

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.MOCK)
@AutoConfigureMockMvc
public class MockEnvTest {

    @Autowired
    private MockMvc mvc;

    @Test
    public void testHello() throws Exception {
        ResultActions ra = mvc.perform(MockMvcRequestBuilders.get(new URI(
                "/hello")));
        MvcResult result = ra.andReturn();
        System.out.println(result.getResponse().getContentAsString());
    }
}
```

为测试类加入@AutoConfigureMockMvc 注解，让其进行 mock 相关的自动配置。在测试方法中，使用 Spring 的 MockMvc 进行模拟测试，向“/hello”发送请求并得到回应。

注意：webEnvironment 属性的默认值是 WebEnvironment.MOCK，只所以在代码清单 2-5 中“多此一举”，是为了展示该配置。

2.3.3 测试业务组件

前面都是针对 Web 容器进行测试，如果不想测试 Web 容器，只是想测试容器中的 bean，则可以只启动 Spring 的容器，请见代码清单 2-6。

代码清单 2-6：codes\02\first-boot\src\main\java\org\crazyit\boot\c2\MyService.java
codes\02\first-boot\src\test\java\org\crazyit\boot\c2\MyServiceTest.java

```
@Service
public class MyService {
```

仅供非商业用途或交流学习使用


```
    public String helloService() {
        return "hello";
    }
}

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class MyServiceTest {

    @Autowired
    private MyService myService;

    @Test
    public void testHello() {
        String result = myService.helloService();
        System.out.println(result);
    }
}
```

在代码清单 2-6 中，新建了一个 MyService 的服务类，MyServiceTest 会对该类进行测试，直接在测试类中注入 MyService 的实例。注意，SpringBootTest 的 webEnvironment 属性被设置为 NONE，因而 Web 容器将不会被启动。

2.3.4 模拟业务组件

在实际应用中，我们的程序可能会操作数据库，也有可能调用第三方接口，为了不让这些外部的不稳定因素影响单元测试的运行结果，可以使用 mock 来模拟某些组件的返回结果，确保被测试组件代码的健壮性。代码清单 2-7 显示了两个业务组件。

代码清单 2-7：codes\02\first-boot\src\main\java\org\crazyit\boot\c2\MainService.java
codes\02\first-boot\src\main\java\org\crazyit\boot\c2\RemoteService.java

```
@Service
public class RemoteService {

    public String call() {
        return "hello";
    }
}

@Service
public class MainService {

    @Autowired
    private RemoteService remoteService;

    public void mainService() {
        System.out.println("这是需要测试的业务方法");
        String result = remoteService.call();
        System.out.println("调用结果：" + result);
    }
}
```

RemoteService 的 call 方法在正常情况下会返回 hello 字符串，MainService 中的 mainService 方法会调用 call 方法。假设 call 方法无法正常运行，为了能测试 MainService，我们需要模拟

仅供非商业用途或交流学习使用

call 方法的返回结果。代码清单 2-8 为 MainService 的测试方法。

代码清单 2-8：codes\02\first-boot\src\test\java\org\crazyit\boot\c2\MockTest.java

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MockTest {

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private MainService mainService;

    @Test
    public void testMainService() {
        // 模拟 remoteService 的 call 方法返回 angus
        BDDMockito.given(this.remoteService.call()).willReturn("angus");
        mainService.mainService();
    }
}
```

在测试类中，使用 MockBean 来修饰需要模拟的组件，在测试方法中使用了 Mockito 的 API 来模拟 remoteService 的 call 方法返回。在模拟中这个方法被调用后，将会返回“angus”字符串，运行代码清单 2-8，输出结果如下：

```
这是需要测试的业务方法
调用结果：angus
```

根据结果可知，RemoteService 的 call 方法被成功模拟。

这一节，简单介绍了如何在 Spring Boot 中进行单元测试，本节的知识基本上能满足大部分的需求，由于篇幅所限，在此不展开讨论。我们下面介绍如何使用 Spring Boot 来发布和调用 REST 服务。

2.4 发布与调用 REST 服务

在系统间进行通信，很多系统都会选择 SOAP 协议，随着 REST 的兴起，现在很多系统在发布与调用 Web Service 时，都首选 REST。这一节，我们介绍如何在 Spring Boot 中发布和调用 REST 服务。

2.4.1 REST

REST 是英文 Representational State Transfer 的缩写，一般翻译为“表述性状态转移”，是 Roy Thomas Fielding 博士在他的论文“Architectural Styles and the Design of Network-based Software Architectures”中提出的一个术语。REST 本身只是分布式系统设计中的一种架构风格，并不是某种标准或者规范，而是一种轻量级的基于 HTTP 协议的 Web Service 风格。从另外一个角度看，REST 更像是一种设计原则，更可以将其理解为一种思想。

2.4.2 发布 REST 服务

在 Spring Boot 中发布 REST 服务非常简单，只需要在控制器中使用@RestController 即可。下面我们来看一个示例。新建一个 rest-server 的 Maven 项目，加入“spring-boot-starter-web”

仅供非商业用途或交流学习使用

依赖，将启动类和控制器写入同一个类中，请见代码清单 2-9。

代码清单 2-9：codes\02\rest-server\src\main\java\org\crazyit\boot\c2\RestApp.java

```
@SpringBootApplication
@RestController
public class RestApp {

    public static void main(String[] args) {
        SpringApplication.run(RestApp.class, args);
    }

    @GetMapping(value = "/person/{name}", produces = MediaType.APPLICATION_JSON_VALUE)
    public Person person(@PathVariable String name) {
        Person p = new Person();
        p.setName(name);
        p.setAge(33);
        return p;
    }

    static class Person {
        String name;
        Integer age;
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public Integer getAge() {
            return age;
        }
        public void setAge(Integer age) {
            this.age = age;
        }

    }
}
```

在代码清单 2-9 中，发布了一个“/person/name”的服务，调用该服务后，会返回一个 Person 实例的 JSON 字符串，该服务对应的方法使用了组合注解@GetMapping，该注解的作用相当于@RequestMapping(method = RequestMethod.GET)。运行代码清单 2-9，在浏览器中访问：http://localhost:8080/person/angus，则返回以下信息：{"name":"angus","age":33}。

很简单的一个注解就帮我们完成了发布 REST 服务的工作，这再一次展示了 Spring Boot 的便捷。如果不使用 Spring Boot，估计你还要为寻找依赖包而疲于奔命。

2.4.3 使用 RestTemplate 调用服务

下面，我们使用 Spring 的 RestTemplate 来调用服务。RestTemplate 是 Spring Framework 的一个类，其主要用来调用 REST 服务，它提供了拦截器机制，我们可以对它进行个性化定制。另外，在 Spring Cloud 中也可以使用 RestTemplate 来调用服务，而且还可以实现负载均衡的功能，有兴趣的朋友可参考笔者的另外一本书《疯狂 Spring Cloud 微服务架构实战》。

我们来看一个例子。

仅供非商业用途或交流学习使用

新建一个 rest-client 的 Maven 项目，加入“spring-boot-starter-web”与“spring-boot-starter-test”的依赖，新建一个最普通的 main 方法，直接调用前面的服务，请见代码清单 2-10。

代码清单 2-10：codes\02\rest-client\src\main\java\org\crazyit\boot\c2\RestTemplateMain.java

```
/**
 * 在 main 方法中使用 RestTemplate
 * @author 杨恩雄
 *
 */
public class RestTemplateMain {

    public static void main(String[] args) {
        RestTemplate tpl = new RestTemplate();
        Person p = tpl.getForObject("http://localhost:8080/person/angus",
Person.class);
        System.out.println(p.getName() + "---" + p.getAge());
    }
}
```

在 main 方法中，直接创建 RestTemplate 的实例并调用服务，操作非常简单。如果想在 Spring 的 bean 里面使用 RestTemplate，则可以使用 RestTemplateBuilder，请见代码清单 2-11。

代码清单 2-11：codes\02\rest-client\src\main\java\org\crazyit\boot\c2\MyService.java

```
@Service
public class MyService {

    @Autowired
    private RestTemplateBuilder builder;

    @Bean
    public RestTemplate restTemplate() {
        return builder.rootUri("http://localhost:8080").build();
    }

    /**
     * 使用 RestTemplateBuilder 来创建 template
     */
    public Person useBuilder() {
        Person p = restTemplate().getForObject("/person/angus", Person.class);
        return p;
    }
}
```

在我们自己的 bean 里面注入 RestTemplateBuilder，创建一个 RestTemplate 的 bean。在创建 RestTemplate 实例时，使用 RestTemplateBuilder 的 rootUri 方法设置访问的 URI。除了 rootUri 方法外，RestTemplateBuilder 还提供了很多方法用于设置 RestTemplate，在此不再赘述。接下来，编写一个单元测试类，来测试我们这个 MyService 的 bean，请见代码清单 2-12。

代码清单 2-12：codes\02\rest-client\src\test\java\org\crazyit\boot\c2\MyServiceTest.java

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class MyServiceTest {

    @Autowired
    private MyService myService;
```

仅供非商业用途或交流学习使用


```
    @Test
    public void testUserTemplate() {
        Person p = myService.useBuilder();
        System.out.println(p.getName() + "===" + p.getAge());
    }
}
```

与前面的单元测试类似，直接注入 MyService 即可。

注意：在运行单元测试时，项目中一定要有 Spring Boot 的启动类，否则会得到以下异常：

java.lang.IllegalStateException: Unable to find a @SpringBootConfiguration, you need to use @ContextConfiguration or @SpringBootTest(classes=...) with your test

Spring 的 RestTemplate 只是众多 REST 客户端中的一个。接下来，我们介绍另外一个 REST 客户端 Feign。

2.4.4 使用 Feign 调用服务

Feign 是 Github 上的一个开源项目，其目的是简化 Web Service 客户端的开发。Spring Cloud 项目将 Feign 整合进来，让其作为 REST 客户端。这一节，我们来了解如何使用 Feign 框架调用 REST 服务。在 rest-client 项目中加入以下依赖：

```
        <dependency>
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-core</artifactId>
            <version>9.5.0</version>
        </dependency>
        <dependency>
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-gson</artifactId>
            <version>9.5.0</version>
        </dependency>
```

新建 PersonClient 接口，请见代码清单 2-13。

代码清单 2-13：codes\02\rest-client\src\main\java\org\crazyit\boot\c2\feign\PersonClient.java

```
/**
 * Feign 客户端接口
 * @author 杨恩雄
 */
public interface PersonClient {

    @RequestLine("GET /person/{name}")
    Person getPerson(@Param("name") String name);

}
```

在接口中，使用了@RequestLine 和@Param 注解，这两个注解是 Feign 的注解。使用注解修饰后，getPerson 方法被调用，然后使用 HTTP 的 GET 方法向“/person/name”服务发送请求。接下来编写客户端运行类，请见代码清单 2-14。

代码清单 2-14：codes\02\rest-client\src\main\java\org\crazyit\boot\c2\feign\FeignMain.java

```
public class FeignMain {
```

仅供非商业用途或交流学习使用


```
    public static void main(String[] args) {
        // 调用 Hello 接口
        PersonClient pc = Feign.builder()
                .decoder(new GsonDecoder())
                .target(PersonClient.class, "http://localhost:8080/");
        Person p = pc.getPerson("angus");
        System.out.println(p.getName() + "---" + p.getAge());
    }
}
```

在代码清单 2-14 中，使用 Feign 来创建 PersonClient 接口的实例，最后通过调用接口方法来访问服务。熟悉 AOP 的朋友可能已经猜到，Feign 实际上帮助我们动态生成了代理类，Feign 使用的是 JDK 的动态代理，代理类会将请求的信息封装，最终使用 java.netHttpURLConnection 来发送 HTTP 请求。如果想将这里的 PersonClient 作为 bean 放到 Spring 容器中，则可以添加一个创建该实例的方法：

```
@Bean
public PersonClient personClient() {
    return Feign.builder()
            .decoder(new GsonDecoder())
            .target(PersonClient.class, "http://localhost:8080/");
}
```

使用 Feign 来调用 REST 服务，使人感觉更加面向对象了。除了 RestTemplate 和 Feign 之外，还可以使用诸如 Restlet、CXF 等框架来调用 REST 服务，在此不再赘述。

2.5 本章小结

本章主要运行了第一个 Spring Boot 程序，通过这个程序，大家可以了解什么是 Spring Boot，在此过程中，大家应该也能感受到 Spring Boot 的便捷。除了这个简单的 Spring Boot 程序外，还介绍了如何在 Spring Boot 环境中运行单元测试，包括对 Web 应用的测试、对 Spring 组件的模拟测试。在本章的最后，介绍了如何在 Spring Boot 中发布和调用 REST 服务，其中重点介绍了 RestTemplate 和 Feign 框架。本章作为 Spring Boot 入门的第一章，涉及的知识较为简单，接下来的章节，我们会继续学习 Spring Boot。

仅供非商业用途或交流学习使用

CHAPTER

3

第3章 Spring Boot 配置

本章要点

- 配置文件
- Web 容器配置
- Java Web 配置
- 自定义配置

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

前面我们使用 Spring Boot 构建了一个 Web 应用。细心的朋友会发现，这个 Web 应用并没有 web.xml 文件，也没有其他配置文件，那么如果要对 Web 应用进行配置，该如何做呢？如果使用自定义配置，又如何实现呢？这一章将向大家详细讲解 Spring Boot 的配置，包括 Spring Boot 的配置文件、容器配置、Web 配置以及自定义配置。

3.1 配置文件

Spring Boot 会按顺序读取各种配置，例如命令行参数、系统参数等。这一节我们讲述配置文件的读取。新建一个名称为“config-file-test”的 Web 项目，加入以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

对配置文件的测试，会在项目中进行，其只是一个使用 Spring Boot 构建的 Web 项目。

3.1.1 默认配置文件

Spring Boot 会读取名称为 application.properties 或者 application.yml 的文件。在默认情况下，会按照以下顺序读取 application.properties（yml）配置文件（如图 3-1 所示）：

1. 项目根目录的 config 目录
2. 项目根目录
3. 项目 classpath 下的 config 目录
4. 项目 classpath 根目录

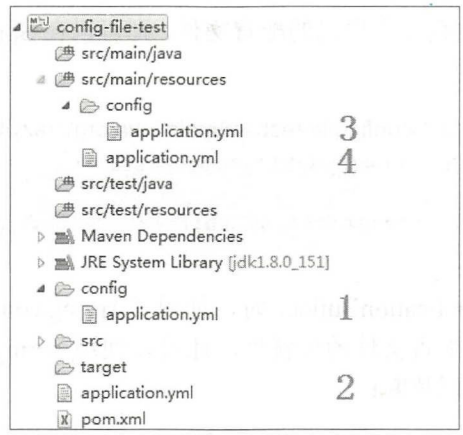


图 3-1 配置文件读取顺序

既然 Spring Boot 会按顺序读取这 4 份配置文件，那么如果同一个配置项出现在多份配置文件中，后读取的值是否会覆盖前面读取的值呢？经笔者测试，是不会覆盖的。我们在 4 份 application.yml 文件里面，加入测试的配置项：

```
user:
  test:
    name: 1  # 按读取的顺序来配置属性值
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

加入的配置项的 key 为 user.test.name，属性值分别是 4 份配置文件的读取顺序。编写启动类和控制器，读取这个配置项，见代码清单 3-1。

代码清单 3-1：codes\03\3.1\config-file-test\src\main\java\org\crazyit\boot\FileApp.java

```
@SpringBootApplication
@RestController
public class FileApp {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FileApp.class).run(args);
    }

    @Autowired
    private Environment env;

    @GetMapping("/ua")
    public String getProp() {
        System.out.println(env.getProperty("user.test.name"));
        return "";
    }
}
```

代码清单 3-1 将应用启动类和控制器放到一起，并注入了 Environment 实例，这是 Spring 框架的环境类，可以使用它来读取配置项。运行 FileApp 后，打开浏览器访问 http://localhost:8080/ua，可以看到控制台输出为 1，这表示 config-file-test/config/application.yml（第 1 顺位的配置文件）的 user.test.name 属性被读取，其他配置文件的 user.test.name 属性值并没有覆盖第 1 顺位配置文件的属性值。

3.1.2 指定配置文件

如果想让 Spring Boot 加载我们自己的配置文件，则可以在 Spring Boot 的启动命令中加入参数，见代码清单 3-2。

代码清单 3-2：codes\03\3.1\config-file-test\src\main\java\org\crazyit\boot\SpecFileApp.java

```
new SpringApplicationBuilder(SpecFileApp.class)
    .properties(
        "spring.config.location=classpath:/test-folder/my-config.properties")
    .run(args);
```

启动类在配置 SpringApplicationBuilder 时，使用了 spring.config.location 属性来指定配置文件的位置。除了可以指定配置文件的位置外，还可以使用 spring.config.name 属性来指定配置文件的名称，例如以下代码片断：

```
new SpringApplicationBuilder(FileName.class).properties(
    "spring.config.name=abc").run(args);
```

以上代码片断指定了配置文件的名称为 abc，那么 Spring Boot 将会到 classpath 下寻找 abc.properties（.yml）文件。一般情况下我们不需要特别指定配置文件，笔者习惯于在项目的 src/main/resources 下使用 application.properties（.yml）文件来进行配置。

3.1.3 yml 文件

YAML 语言可使用一种方便的格式进行数据配置，通过配置分层、缩进等格式，来增强

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

配置文件的可读性。使用 YAML 语言的配置文件以“.yml”作为后缀，代码清单 3-3 即是一个 yml 配置文件示例。

代码清单 3-3：codes\03\3.1\config-file-test\src\main\resources\my-config.yml

```
jdbc:
  user:
    root
  passwd:
    123456
  driver:
    com.mysql.jdbc.Driver
```

需要特别注意的是，每一行配置的缩进要使用空格，不能使用 Tab 键进行缩进。代码清单 3-3 对应的 properties 文件内容如下：

```
jdbc.user=root
jdbc.passwd=123456
jdbc.driver=com.mysql.jdbc.Driver
```

注意：如无特别说明，本书的大部分项目会使用 yml 文件进行配置。

3.1.4 使用 profile 指定配置

在实际应用中，有时我们可能需要根据特定的环境来激活不同的配置，针对这样的情况，可以考虑使用 profile 特性。在代码清单 3-4 的 yml 文件中定义了两个 profile。

代码清单 3-4：codes\03\3.1\config-file-test\src\main\resources\test-profile.yml

```
spring:
  profiles: mysql
jdbc:
  driver:
    com.mysql.jdbc.Driver
---
spring:
  profiles: oracle
jdbc:
  driver:
    oracle.jdbc.driver.OracleDriver
```

两个 profile 之间使用“---”进行分隔。接下来，编写启动类来设置激活的 profile，见代码清单 3-5。

代码清单 3-5：codes\03\3.1\config-file-test\src\main\java\org\crazyit\boot\TestProfile.java

```
@SpringBootApplication
@RestController
public class TestProfile {

    public static void main(String[] args) {
        // 读取控制台输入
        Scanner scan = new Scanner(System.in);
        String profile = scan.nextLine();
        new SpringApplicationBuilder(
                TestProfile.class)
                .properties(
                        "spring.config.location=classpath:/test-profile.yml")
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
                .profiles(profile).run(args);
    }

    @Autowired
    private Environment env;

    @GetMapping("/tp")
    public String getProp() {
        System.out.println(env.getProperty("jdbc.driver"));
        return "";
    }
}
```

在代码清单 3-5 中，先读取控制台的输入，再调用 profiles 方法来设置需要激活的 profile。在启动应用时，在控制台中输入 mysql，打开浏览器访问 http://localhost:8080/tp，可以看到控制台输出为：com.mysql.jdbc.Driver。除了使用 profiles 方法外，还可以通过设置 spring.profiles.active 属性来设置激活的 profile，效果与使用 profiles 方法一致。

除此之外，还可以通过配置文件的名称来设置 profile，例如在 classpath 下有两份文件：application-mysql.yml 和 application-oracle.yml，这实际上就等于定义了 mysql 和 oracle 两个 profile。在启动应用时，只需要输入相应的文件名称即可激活它，而且在配置文件中，也不再需要设置 spring.profiles 属性。

3.2 Web 配置

在 Servlet 3 规范出来之后，Web 项目可以不使用 web.xml 配置文件。该规范提供了相关注解，可以用来实现 web.xml 的配置工作，例如 Servlet、Listener、Filter 等都可以使用注解进行配置。这一节将向大家介绍相关的 Web 配置。

3.2.1 Servlet 配置

新建一个名称为“web-test”的 Maven 项目，在该项目中只使用“spring-boot-starter-web”依赖。新建启动类，请见代码清单 3-6。

代码清单 3-6：codes\03\3.2\web-test\src\main\java\org\crazyit\boot\c3\WebApp.java

```
@SpringBootApplication
@ServletComponentScan
public class WebApp {

    public static void main(String[] args) {
        SpringApplication.run(WebApp.class, args);
    }
}
```

在启动类中，除了包含原来的 SpringBootApplication 注解外，还加入了 ServletComponentScan 注解，这个注解主要用于扫描源码包中的 Servlet 组件，包括使用@WebServlet、@WebFilter 和@WebListener 进行修饰的类。接下来，新建一个 Servlet，使用@WebServlet 进行修饰，请见代码清单 3-7。

代码清单 3-7：odes\03\3.2\web-test\src\main\java\org\crazyit\boot\c3\web\MyServlet.java

```
@WebServlet(value = "/servlet")
public class MyServlet extends HttpServlet {
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    public MyServlet() {
        System.out.println("my servlet class");
    }

    protected void service(HttpServletRequest arg0, HttpServletResponse arg1)
            throws ServletException, IOException {
        System.out.println("servlet service method");
    }
}
```

代码清单3-7定义了一个MyServlet，在浏览器中访问http://localhost:8080/servlet，则可以看到这个service方法的执行情况。MyServlet依然继承于HttpServlet，只是使用@WebServlet进行了修饰。该注解主要有以下属性。

- name：String类型，配置servlet的名称。
- value：String数组，相当于配置了urlPatterns属性。
- urlPatterns：String数组，用于配置多个url-pattern，配置方式如下：

```
@WebServlet(urlPatterns = {"a", "b"})
```

- loadOnStartup：相当于<load-on-startup>元素。
- initParams：WebInitParam数组，用于设置Servlet的初始化参数，配置方式如下：

```
@WebServlet(value = "/servlet", initParams = { @WebInitParam(name = "paramA", value
= "angus") })
```

- asyncSupported：等同于Servlet的<async-supported>元素，用于配置该Servlet是否支持异步。
- description：等同于<description>元素。
- displayName：等同于<display-name>元素。

3.2.2 Listener配置

监听器（Listener）与Servlet一样，使用@WebListener注解进行修饰后，就可以被ServletComponentScan扫描并注册，请见代码清单3-8。

代码清单3-8：codes\03\3.2\web-test\src\main\java\org\crazyit\boot\c3\web\MyListener.java

```
@WebListener
public class MyListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("context init");
    }

    public void contextDestroyed(ServletContextEvent sce) {

    }
}
```

@WebListener注解只有一个value属性，用来描述这个Listener，其使用方法较为简单，在此不再赘述。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

3.2.3 Filter 配置

使用@WebFliter 来修饰一个过滤器（Filter），请见代码清单 3-9。

代码清单 3-9：codes\03\3.2\web-test\src\main\java\org\crazyit\boot\c3\web\MyFilter.java

```
@WebFilter(value = "/filter/*")
public class MyFilter implements Filter {

    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("my filter init");
    }

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        System.out.println("doFilter method");
        chain.doFilter(request, response);
    }

    public void destroy() {

    }
}
```

@WebFilter 注解主要有以下属性。

- filterName：String 类型，用于指定过滤器名称。
- value：String 数组，相当于配置了 urlPatterns。
- urlPatterns：String 数组，用于配置多个 url-pattern，等同于<url-pattern>元素。
- servletNames：配置过滤器将会在哪些 Servlet 中使用。
- dispatcherTypes：DispatcherType 数组，相当于配置了<filter-mapping>的<dispatcher>元素，其取值可以为 FORWARD、INCLUDE、REQUEST、ASYNC、ERROR。

除了以上几个属性外，还有 initParams、asyncSupported、description 与 displayName 属性，它们的使用方法与@WebServlet 一致。

通过上面的例子可以知道，Servlet、Listener 与 Filter 这 3 个注解基本上与 web.xml 中的配置一样，只是将 XML 代码变为注解而已，使用方法较为简单。再次提醒，要想让这 3 个注解生效，则需要使用 ServletComponentScan 注解。

3.2.4 构建可部署的 war 包

前面的所有示例都使用了内嵌的 Tomcat 服务器，这些项目最终可被打成 jar 包来运行，每个 jar 包都可以被看作一个独立的 Web 服务器。对于传统的 Web 开发，一般会将 Web 应用打成一个 war 包，然后将其部署到 Web 服务器中运行。Spring Boot 也支持传统的部署模式。新建一个 Maven 项目，其 pom.xml 文件内容如代码清单 3-10 所示。

代码清单 3-10：codes\03\3.2\war-test\pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    <groupId>org.crazyit.boot.c3</groupId>
    <artifactId>war-test</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
    <build>
        <finalName>war-test</finalName>
    </build>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.1.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
    </dependencies>

</project>
```

注意 pom.xml 文件中的粗体字部分，该项目最终会被打成一个名称为“war-test”的 war 包，其所依赖的“spring-boot-starter-tomcat”项目，会由 war 所在的容器提供。接下来，编写启动类与测试用的控制器，请见代码清单 3-11。

代码清单 3-11：codes\03\3.2\war-test\src\main\java\org\crazyit\boot\c3\WarApp.java

```
@SpringBootApplication
@RestController
public class WarApp extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(WarApp.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(WarApp.class, args);
    }

    @GetMapping("/hello")
    public String hello() {
        return "hello";
    }
}
```

启动类继承自 SpringBootServletInitializer 类，这里需要重写父类的 configure 方法。SpringBootServletInitializer 类将会调用 configure 方法来得到一个 SpringApplicationBuilder 实例。根据名称可知，这个 SpringApplicationBuilder 会帮我们创建 Spring 上下文，实际上它会帮我

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

们创建 WebApplicationContext 实例，其在创建 Spring 上下文的过程中，会查找使用了@Configuration 注解的配置类，然后对它们进行初始化。

对于启动类，我们可以这样理解：在 Eclipse 运行 main 方法时，会使用原来的方式（执行 jar 包的方式）来启动 Web 应用（含服务器），如果将项目打成一个 war 包放到 Web 服务器上运行，就会创建 WebApplicationContext，在创建的过程中，会调用 configure 方法。另外，父类调用一个方法来实现部分工作，但该方法需要由继承的子类来实现，这种模式是 GoF 设计模式的一种：模板方法。

编写完启动类和控制器之后，使用命令行工具，进入项目的根目录（codes\03\3.2\ war-test），输入 Maven 命令对项目进行构建：mvn clean package。执行成功后，到 codes\03\3.2\war-test\target 目录，可以看到 war-test.war，将该包复制到 Tomcat 的 webapps 下。启动 Tomcat 后，访问：http://localhost:8080/war-test/hello，在正常情况下可以看到结果，笔者在 Tomcat 8 中测试成功，建议使用 7.0 以上版本的 Tomcat。

3.2.5 JSP 配置

JSP 是传统 Web 应用最常使用的表现层技术，Spring Boot 的官方文档不建议在可执行的 jar 包中使用 JSP，但我们可以在 war 包里面使用它。新建一个 Maven 项目，在配置中将其打成一个 war 包（packaging 元素），在 pom.xml 文件中加入相应的依赖，请见代码清单 3-12。

代码清单 3-12：codes\03\3.2\jsp-test\pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
    </dependency>
</dependencies>
```

前面两个依赖我们之前介绍过，这里重点来看下“tomcat-embed-jasper”，这个包主要用于编译 JSP，它同样由容器提供。另外，本例将会使用 JSTL 标签，因此加入 JSTL 的依赖。在 src/main/resources 下新建 application.yml，添加以下内容：

```
spring:
  mvc:
    view:
      prefix: /pages/
      suffix: .jsp
```

仅供非商业用途或交流学习使用

这段内容用于配置 Spring MVC 视图，本例默认会将 JSP 文件放到 src/main/webapp/ pages 目录下。新建启动类与控制器，请见代码清单 3-13。

代码清单 3-13：codes\03\3.2\jsp-test\src\main\java\org\crazyit\boot\c3\JspApp.java

```
@SpringBootApplication
@Controller
public class JspApp extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(JspApp.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(JspApp.class, args);
    }

    @GetMapping("/hello")
    public String hello(HttpServletRequest request) {
        request.setAttribute("name", "angus");
        return "hello";
    }
}
```

控制器的 hello 方法会设置名称为 name 的参数，最后跳转到 pages 下面的 hello.jsp。新建 hello.jsp，内容请见代码清单 3-14。

代码清单 3-14：codes\03\3.2\jsp-test\src\main\webapp\pages\hello.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<c:set var="ctx" value="${pageContext.request.contextPath}" />
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <link rel="stylesheet" href="${ctx}/css/test.css" type="text/css" />
        <script type="text/javascript" src="${ctx}/js/test.js"></script>
        <title></title>
    </head>
    <body>
        <div onClick="myClick()" class="main_text">Hello ${name}</div>
    </body>
</html>
```

这是一个最简单的 JSP 文件，在开头声明了 JSTL 的标签库与 ctx 变量，该变量用于定位 CSS 与 JS 资源。在 JSP 中使用了一个 div 元素来显示 name 参数。本例的项目结构如图 3-2 所示。

在 Eclipse 中运行 JspApp 类，打开浏览器访问：http://localhost:8080/hello，则可以看到 JSP 页面的输出。同样地，可以使用命令行工具进入 jsp-test 目录，执行 mvn clean package 命令将项目打成一个 war 包，放到另外的 Tomcat 容器下运行。在浏览器中访问：http://localhost:8080/jsp-test/hello，可以看到同样的结果。

仅供非商业用途或交流学习使用

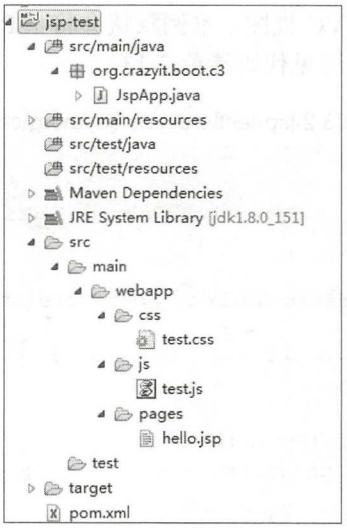


图 3-2 项目结构

3.3 其他配置

这一节将会讲述一些较为常用的配置，例如压缩、SSL 等。

3.3.1 服务器常用配置

Spring Boot 通用的配置项大约有一千多个，这些配置项大多与某个特定框架相关。在日常的使用中，还会涉及一些服务器的配置，这些配置基本上都以“server”作为前缀。下面我们介绍一些较为常用的服务器配置。

- server.port：配置服务器的 HTTP 端口，默认为 8080。
- server.context-path：应用的访问路径。
- server.connection-timeout：HTTP 请求的连接超时时间，单位为毫秒，如果不设置，将会使用特定服务器的超时时间，设置为-1 表示不会超时。
- server.session.timeout：session 的超时时间，单位为秒。

如果读者需要了解更多的配置，可以到 Spring Boot 的官方文档查看，也可以到本书的 codes\03 目录下查找 SpringProperties.properties 文件，笔者已经将全部配置复制到该文件中了。

3.3.2 响应压缩

Spring Boot 默认支持 Gzip 压缩，使用该压缩比较简单，只需要在 application.yml 里面配置即可。新建一个 Maven 项目，加入“spring-boot-starter-web”的依赖。项目的启动类和控制器代码见代码清单 3-15。

代码清单 3-15：compression-test\src\main\java\org\crazyit\boot\c3\ComApp.java

```
@SpringBootApplication
@Controller
public class ComApp {

    public static void main(String[] args) {
```

仅供非商业用途或交流学习使用


```
        SpringApplication.run(ComApp.class, args);
    }

    @RequestMapping(path = "/hello")
    @ResponseBody
    public String hello() {
        return "hello world";
    }
}
```

在控制器中，只提供了一个“/hello”服务，返回“hello world”字符串。接下来配置application.yml文件，请见代码清单3-16。

代码清单3-16：compression-test\src\main\resources\application.yml

```
server:
  compression:
    enabled: true
    min-response-size: 1
    mime-types:
text/html,text/xml,text/plain,text/css,text/javascript,application/javascript
```

其中enabled配置是否打开压缩开关；min-response-size配置压缩阀值，单位是字节，默认值为2048，本例的配置较为极端，当响应的数据大于1字节时，就会进行压缩；mime-types属性用于配置需要压缩的报文类型，这里使用默认值，如果需要对JSON进行压缩，加上“application/json”即可。

编写请求客户端，调用hello服务，请求客户端可以使用CXF、Feign等框架。为了简单起见，我们直接使用RestTemplate，请见代码清单3-17。

代码清单3-17：codes\03\3.3\compression-test\src\test\java\org\crazyit\boot\c3\TestMain.java

```
public class TestMain {

    public static void main(String[] args) throws Exception {
        // 创建RestTemplate实例，也可以在单元测试中使用TestRestTemplate
        RestTemplate restTemplate = new RestTemplate();
        HttpHeaders requestHeaders = new HttpHeaders();
        // 设置编码为gzip
        requestHeaders.set("Accept-Encoding", "gzip");
        HttpEntity<?> requestEntity = new HttpEntity<Object>(requestHeaders);
        // 调用hello服务
        ResponseEntity<byte[]> entity = restTemplate.exchange(
                "http://localhost:8080/hello", HttpMethod.GET, requestEntity,
                byte[].class);
        // 读取返回的内容
        GZIPInputStream inflater = new GZIPInputStream(
                new ByteArrayInputStream(entity.getBody()));
        String s = StreamUtils.copyToString(inflater, Charset.forName("UTF-8"));
        System.out.println(s);
    }
}
```

在客户端代码中，直接使用RestTemplate调用“/hello”服务。注意，要在请求头中编码，获取响应后会得到byte[]数组，使用GZIPInputStream将byte[]数组转换为字符再输出。运行启动类（ComApp），再运行TestMain类，则可以看到客户端返回“hello world”字符串。

3.3.3 SSL 配置

我们还可以对 Web 应用内嵌的服务器进行配置，让其支持 HTTPS。使用 keytool 工具生成我们自己的证书，命令如下：

```
keytool -genkeypair -alias "sslTestKey" -keyalg "RSA" -keystore "D:\sslTest.keystore"
```

在此过程中会提示我们输入密码等信息。接下来，新建一个 Maven 项目，添加“spring-boot-starter-web”依赖。在该项目中只发布一个“/hello”服务，返回“hello”字符串。启动类与控制器的代码与前面的项目一致，在此不再赘述，代码请见“codes\03\3.3\ssl-test\src\main\java\org\crazyit\boot\c3\SslApp.java”。

将前面生成的 sslTest.keystore 文件复制到 src/main/resources 下，并在项目的 application.yml 中进行 SSL 的配置，请见代码清单 3-18。

代码清单 3-18：codes\03\3.3\ssl-test\src\main\resources\application.yml

```
server:
  port: 8443
  ssl:
    key-store: classpath:sslTest.keystore
    key-store-password: 123456
    key-password: 123456
```

属性 key-store 用于指定 keystore 文件的位置，key-store-password 用于配置密钥库的密码，key-password 用于配置密钥的密码。启动项目，在浏览器中访问：https://localhost: 8443/hello，则可以看到具体结果。

3.3.4 使用其他服务器

除了可以使用内置的 Tomcat 服务器外，还可以使用 Jetty 及 JBoss Undertow 服务器。使用这两个服务器较为简单，只需要修改 Maven 的脚本文件即可。以 Jetty 为例，项目的 pom.xml 配置请见代码清单 3-19。

代码清单 3-19：codes\03\3.3\use-jetty\pom.xml

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.1.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jetty</artifactId>
```

```
        </dependency>
    </dependencies>
```

在添加“spring-boot-starter-web”依赖时，使用 exclusion 元素将 Tomcat 的依赖排除，再另外加入“spring-boot-starter-jetty”依赖。编写启动类进行测试（本例的运行类为 codes\03\3.3\use-jetty\src\main\java\org\crazyit\boot\c3\JettyApp.java），运行成功后可以看到以下的日志输出：

```
Jetty started on port(s) 8080 (http/1.1)
```

与 Jetty 类似，使用 Undertow 的话，先将 Tomcat 依赖排除，再加入“spring-boot-starter-undertow”依赖，启动后也有类似于 Jetty 的日志输出。如无特别说明，本书的所有示例，都使用 Tomcat 服务器运行。

3.3.5 服务器访问日志

服务器访问日志与服务器日志并不是同一个概念，我们一般在 Tomcat 控制台中看到的日志，是服务器的日志，而服务器访问日志，则会记录服务处理的请求信息，默认情况下，Tomcat 并不会记录访问日志。新建一个 Maven 项目并为其加入“/hello”服务，在配置文件中，对访问日志进行配置，请见代码清单 3-20。

代码清单 3-20：codes\03\3.3\log-test\src\main\resources\application.yml

```
server:
  tomcat:
    basedir: my-tomcat
    accesslog:
      pattern: 'ip: %A, response code: %s, time: %t'
      enabled: true
      directory: crazyit-logs
      buffered: false
```

在代码清单 3-20 的配置项目中，需要先指定 Tomcat 的根目录，本例指定了 my-tomcat 目录，也就是会以 log-test/my-tomcat 作为 Tomcat 根目录。使用 directory 来配置访问日志的存放目录，本例中指定为 my-tomcat/crazyit-logs。属性 accesslog.enabled=true 用于开启日志记录，accesslog.buffered 被设置为 false，表示不进行缓冲，直接将日志记录到文件中。

访问日志以哪种格式进行保存呢？要使用 accesslog.pattern 属性进行配置，可以使用 Tomcat 提供的标识符来表示不同的信息，例如本例中的“%A”表示本地的 IP 地址，%s 表示 HTTP 的响应码，%t 表示含时间的日期格式。除了这里使用的 3 个标识符外，其他标识符可以参看 Tomcat 文档：https://tomcat.apache.org/tomcat-8.0-doc/config/valve.html# Access_Logging。

启动本例的服务器（codes\03\3.3\log-test\src\main\java\org\crazyit\boot\c3\LogApp.java），访问：http://localhost:8080/hello，到 log-test/my-tomcat/crazyit-logs 目录下，则可以看到访问日志已经生成，日志内容如下：

```
ip: 192.168.1.102, response code: 200, time: [07/Nov/2017:13:48:47 +0800]
```

如果读者在实际环境中使用日志分析工具（例如 ELK），则可以结合以上的配置，将日志存放到特定目录，再让日志分析工具按照设置好的格式读取入库。

本节介绍了 Spring Boot 常用的 Server 配置，这些配置项都以“server”开头。本节介绍的配置基本上可以满足需求，如果你想了解更多的 Server 配置，则可以到 codes\03 目录下，打

开 SpringProperties.properties 文件，查找“server.xxxx”格式的配置。

3.3.6 banner 配置

Spring Boot 在启动时，会将 banner 信息显示到控制台中，默认的 banner 如图 3-3 所示。

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.2.RELEASE)
```

图 3-3　默认 banner

我们可以通过配置 application.yml 文件，修改默认的 banner，例如在 classpath 下提供一份 txt 文本文件，然后将 spring.banner.location 属性配置为“classpath:banner.txt”，这样在启动时，就可以将默认的 banner 替换掉。除了文本外，还可以提供图片文件用于显示，图片格式可以为 jpg、png 或 gif，Spring Boot 会将图片转换为 ASCII，以文本的方式将图片显示到控制台中。以疯狂 Java 论坛的 logo 为例，使用图片作为 banner，如图 3-4 所示为 logo 原图。



图 3-4　疯狂 Java 论坛 logo

将其配置为 banner，application.yml 的内容如下：

```
spring:
  banner:
    charset: UTF-8
    location: classpath:banner.txt
    image:
      location: classpath:logo.jpg
      invert: false
```

启动 Spring Boot，控制台输出如图 3-5 所示。

在 Spring Boot 中配置 banner，可以在 application.yml 中配置以下几个属性。

- spring.banner.charset：如果 banner 的文本文件中有 UTF-8 以外的编码，则需要配置该项。
- spring.banner.location：用于指定 banner 的文本文件位置。
- spring.banner.image.location：指定 banner 图片的位置。
- spring.banner.image.width：banner 图片经转换后的字符长度，默认为 76。
- spring.banner.image.height：图片经转换后的字符高度。
- spring.banner.image.margin：设置图片显示的边距，默认为 2。
- spring.banner.image.invert：配置为 true，则将图片进行转换显示，以适应深色的终端风格。

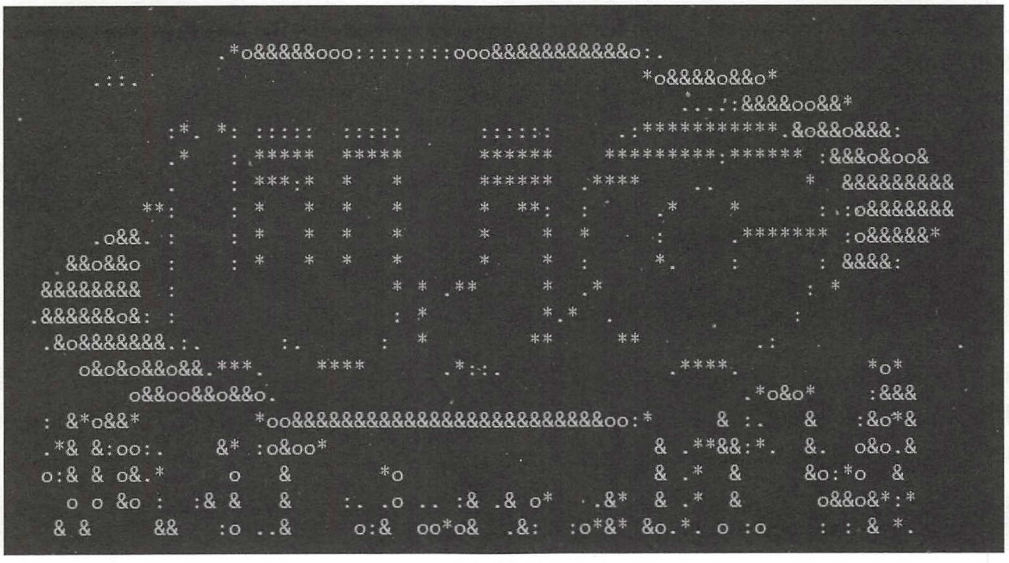


图 3-5 使用图片配置 banner

还可以将特定的变量传递到 banner 文本文件中。在文本文件中可以使用以下的变量。

- ${application.version}：读取 MANIFEST.MF 文件里面的 Implementation-Version，Spring Boot在打包时，会将pom.xml文件中的version值设置为Implementation-Version。
- ${application.formatted-version}：格式化过的 application.version 值，例如 application.version 值为 1.0，则格式化后值为(v1.0)。
- ${spring-boot.version}：Spring Boot 的版本，如 2.0.1.RELEASE。
- ${spring-boot.formatted-version}：格式化后的 Spring Boot 版本，例如(v2.0.2.RELEASE)。
- ${application.title}：应用名称，读取 MANIFEST.MF 里面的 Implementation-Title，打包后使用 pom.xml 的 artifactId。

除了以上的变量外，还可以使用 AnsiColor.XXX、AnsiBackground.XXX、Ansi.XXX 与 AnsiStyle.XXX 来控制输出的 banner 样式。

3.4 自定义配置

我们可以直接在application.yml文件中定义我们自己的配置，问题是如何使用定义的配置。在前面的章节中，我们使用过 Environment 来读取配置，如果属性较多，这种方式会带来混乱。Spring 提供了@Value 注解，从而可以让我们的业务代码去读取配置，而 Spring Boot 提供了一种更优雅的方式来获取配置。这一节我们先回顾 Spring 的@Value 注解，再学习 Spring Boot 的配置使用。

3.4.1 @Value 注解

在使用 Spring 的时候，如果要在程序中使用属性，则可以定义一个属性类，再使用@Value 注解修饰相应的属性即可，请见代码清单 3-21。

代码清单 3-21：codes\03\3.3\prop-test\src\main\java\org\crazyi\boot\c3\value\ValueProp.java

```
@Component
public class ValueProp {

    @Value("${jdbc.url}")
    private String jdbcUrl;

    public String getJdbcUrl() {
        return jdbcUrl;
    }
}
```

可将代码清单 3-21 中的 ValueProp 类称为属性映射类，该类将会注册到 Spring 容器中。在使用@Value 注解时，需要设置属性值的 key，本例中的 key 为 jdbc.url，对应到 application.yml 文件中，应当这样配置：

```
jdbc:
  url: value
```

接下来，编写控制器，来使用 ValueProp 的属性，请见代码清单 3-22。

代码清单 3-22：codes\03\3.3\prop-test\src\main\java\org\crazyi\boot\c3\PropController.java

```
@RestController
public class PropController {

    @Autowired
    private ValueProp valueProp;

    @GetMapping(value = "/value", produces = MediaType.APPLICATION_JSON_VALUE)
    public ValueProp value() {
        return valueProp;
    }
}
```

直接在控制器中注入 ValueProp 实例，value 方法返回 ValueProp 实例。启动服务器后（本例为 codes\03\3.3\prop-test\src\main\java\org\crazyi\boot\c3\PropApp.java），在浏览器中输入：http://localhost:8080/value，可以看到浏览器输出如下：

```
{"jdbcUrl":"localhost:3306"}
```

使用@Value 注解，使得属性的获取更加面向对象。使用属性时也不需要关心属性的 key，但@Value 注解会带来一个问题，即在映射属性时，需要传入“jdbc.url”，如果 jdbc 下面有多个属性，就会造成代码的冗余。对这种情况，可以使用 Spring Boot 的配置。

3.4.2 定义属性映射对象

假设我们在配置文件中定义一些 JDBC 的属性，请见代码清单 3-23。

代码清单 3-23：codes\03\3.3\prop-test\src\main\resources\application.yml

```
jdbc:
  url: localhost:3306
  user: root
  password: 123456
  db:
    name: mysql
```

```
    version: 1.0
    roles:
      - manager
      - client
```

与使用@Value注解类似，新建一个属性映射类，请见代码清单3-24。

代码清单3-24：codes\03\3.3\prop-test\src\main\java\org\crazyi\boot\c3\MyProp.java

```
@ConfigurationProperties(prefix = "jdbc")
public class MyProp {

    private String url;

    private String user;

    private String password;

    private Database db;

    ...省略setter和getter方法

    public static class Database {

        private String name;

        private String version;

        private List<String> roles;

        ...省略setter和getter方法
    }
}
```

在代码清单3-24中，定义了属性映射类MyProp，这个类使用了@ConfigurationProperties注解进行修饰，并且声明了该类的配置前缀为“jdbc”，这样在使用MyProp的getUrl方法时，就可以获取配置文件中的“jdbc.url”属性值。需要注意的是db属性，它是Database类型，Database类型下的roles属性是一个字符串集合，在配置时，需要使用以下格式：

```
jdbc:
  db:
    roles:
      - manager
      - client
```

以上的配置会为Database的roles属性加入manager与client两个常量。以上在Spring Boot中自定义了属性，下面，讲解如何使用这些属性。

3.4.3 使用配置

建立了属性映射类后，我们还要告诉Spring容器，这里有一个自定义的属性映射对象。新建一个配置类，请见代码清单3-25。

代码清单3-25：codes\03\3.3\prop-test\src\main\java\org\crazyi\boot\c3\MyConfig.java

```
@Configuration
@EnableConfigurationProperties(value = MyProp.class)
```

```
public class MyConfig {

}
```

在前面定义属性映射类时，使用了@ConfigurationProperties 注解来修饰，但这还不能让 Spring 容器发现它的存在，还要使用代码清单 3-25 中的@EnableConfigurationProperties 注解，这个注解会让 Spring 容器去注册属性映射对象，本例中的 MyProp 对象，将会作为一个 bean 被放到 Spring 容器中。编写一个控制器使用 MyProp 对象，请见代码清单 3-26。

代码清单 3-26：codes\03\3.3\prop-test\src\main\java\org\crazyi\boot\c3\PropController.java

```
@RestController
public class PropController {

    @Autowired
    private MyProp prop;

    @GetMapping(value = "/hello", produces = MediaType.APPLICATION_JSON_VALUE)
    public MyProp hello() {
        return prop;
    }
}
```

直接在控制器中使用@Autowird 注入一个 MyProp 实例，在 hello 方法中，会把这个 MyProp 实例作为 JSON 字符串返回。启动服务器（PropApp.java），在浏览器中访问：http://localhost:8080/hello，浏览器输出如下：

```
{"url":"localhost:3306","user":"root","password":"123456","db":{"name":"mysql",
"version":"1.0","roles":["manager","client"]}}
```

根据输出结果可知，我们所设计的属性映射对象（MyProp），已经可以被控制器使用。Spring Boot 的这种方式，相对于@Value 注解来说，在 Java 类中可以减少配置的冗余，同时也增强了配置的可读性，使用起来更加简便。

3.4.4 属性自动识别

假设有以下属性映射类：

```
@ConfigurationProperties(prefix = "jdbc")
public class MyProp {

    private String userName;
}
```

前缀是 jdbc，属性为 userName，如果使用了@ConfigurationProperties，那么以下的几种配置都会被识别，并且被设置到 MyProp 类的 userName 中：

- jdbc.userName
- jdbc.user-name
- jdbc.user_name
- JDBC_USER_NAME

属性自动识别，官方文档称其为“relaxed binding”。关于自定义配置就讲解到这里，这里所介绍的配置知识，基本上可以满足实际环境的需要。

3.5 本章小结

本章主要介绍了 Spring Boot 的基础配置，包括最基本的配置文件加载、Web 相关的配置、Server 配置等。学习完本章后，读者可以掌握 Spring Boot 自带的一些常用配置。Spring Boot 的公共配置大约有一千多个，它们都需要结合某些特定框架来使用。如果在实际应用过程中，这些配置无法满足要求，则可以使用自定义配置。在本章最后一节，讲解了在 Spring Boot 中使用自定义配置的方法。通过本章的学习，读者可以初步掌握 Spring Boot 的配置知识。

CHAPTER

4

第4章 Spring Boot 的注解

本章要点

- Spring 的常用注解
- 高级 Spring 注解
- Spring MVC 的注解
- Spring Boot 的条件注解
- Spring Boot 的自动配置

我们在前面章节接触了许多注解，例如启动类的@SpringBootApplication，控制器的@Controller、@RestController，还有与配置相关的一系列注解。在使用 Spring Boot 时，在大部分的情况下，我们都会使用这些注解来实现具体的功能，因此对于一些常用的注解，我们需要掌握。需要注意的是，可能有些朋友更加喜欢使用 XML 进行配置，这一章我们不讨论 XML 与注解模式孰优孰劣，只希望你通过学习本章，为后面章节的学习打下基础。

4.1 Spring 的常用注解

Spring 框架主要包括 IoC 与 AOP，这两大功能都可以使用注解进行配置，本节就以这两大功能为基础，讲解 Spring 的基本注解。新建一个 Maven 项目，在 pom.xml 中加入“spring-boot-starter-web”依赖，这个项目将用于测试。

注意：本节将会直接使用 Spring Boot 的环境来测试 Spring 的注解，省去了使用编码的方式来创建 Spring 容器，这些工作将由 Spring Boot 完成。

4.1.1 bean 定义

使用 XML 的方式配置 Spring，需要告诉 Spring 容器 XML 的位置，然后在 XML 中使用<bean>元素来定义 bean。

查找 XML 文档与配置<bean>元素这两个工作同样可以使用注解来完成。使用@Component、@Service 或者@Configuration 注解来修饰一个 Java 类，这些 Java 类会被 Spring 自动检测并注册到容器中，在类里面使用@Bean 注解修饰的方法，会被作为一个 bean 存放到 Spring 容器中。代码清单 4-1 即使用了@Configuration 和@Bean 注解。

代码清单 4-1：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\bean\MyBean.java
codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\bean\BeanApp.java

```
public class MyBean {

}

@Configuration
public class MyConfig {

    @Bean
    public MyBean getMyBean() {
        return new MyBean();
    }

    @Bean("bean2")
    public MyBean myBean2() {
        return new MyBean();
    }
}
```

在 MyConfig 类中使用了两个@Bean 注解。在使用@Bean 注解时，如果没有传入 bean 的名称，默认会使用方法名作为 bean 的名称。接下来编写 Spring Boot 的启动类和控制器，并在容器中读取这两个 bean，请见代码清单 4-2。

代码清单 4-2：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\bean\BeanApp.java

```
@SpringBootApplication
@RestController
public class BeanApp {

    public static void main(String[] args) {
        SpringApplication.run(BeanApp.class, args);
    }

    @Autowired
    ApplicationContext ctx;

    /**
     * 输出容器中全部的 bean
     * @return
     */
    @GetMapping("/print")
    public String printBeans() {
        // 根据类型获取 bean 的名称
        String[] names = ctx.getBeanNamesForType(MyBean.class);
        for(String name : names) {
            System.out.println(name);
        }
        return "";
    }
}
```

代码清单 4-1 将启动类和控制器放到一起，在控制器中会注入 ApplicationContext 的实例，printBean 方法会通过该实例，获取 MyBean 类型的 bean 名称并输出到控制台。运行代码清单 4-1，在浏览器中访问：http://localhost:8080/print，控制台输出如下：

```
getMyBean
bean2
```

根据结果可知，Spring 容器中存在两个 MyBean 类型的 bean。本例在修饰 MyConfig 时使用了@Configuration 注解，这个注解实际上已经具有了@Component 的功能，一般情况下我们直接使用@Configuration 注解即可。

我们可以换一个角度来理解@Configuration 与@Bean 这两个注解，使用@Configuration 注解修饰的类，我们可以把它看作一个 Spring 的 XML 配置文件，而在类里面使用@Bean 修饰的方法，则可以被看作 XML 文件中的一个<bean>元素。还有一个细节需要注意，使用@Configuration 注解修饰的类，本身也会作为一个 bean 被注册。

4.1.2 依赖注入

想要在一个 bean 中使用另一个 bean，则可以直接在使用者一端，将被调用者的实例注入，例如下面的 XML，在 beanA 中注入 beanB 的实例：

```
<bean id="beanA" class="xxx">
    <property name="beanB" ref="beanB"/>
</bean>
```

使用注解同样可以实现实例的注入，最常用的是@Resource 及@Autowired 这两个注解。其中@Resource 注解是 JSR-250 规范定义的注解，该注解默认会根据名称进行注入，而

@Autowired注解默认会根据类型进行注入。代码清单4-3使用了这两个注解实现依赖注入。

代码清单4-3：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\inject\InjectBean.java
codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\inject\InjectConfig.java

```
public class InjectBean {

    private String id;

    // 省略 setter 和 getter 方法
}

@Configuration
public class InjectConfig {

    @Bean
    public InjectBean myBean1() {
        return new InjectBean("1");
    }

    @Bean
    public InjectBean myBean2() {
        return new InjectBean("2");
    }
}
```

在代码清单4-3中定义了两个名称不同的InjectBean类型的bean，这两个bean都会设置不同的id属性，它们将会被注入到控制器中，请见代码清单4-4。

代码清单4-4：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\inject\InjectApp.java

```
@SpringBootApplication
@RestController
public class InjectApp {

    public static void main(String[] args) {
        SpringApplication.run(InjectApp.class, args);
    }

    // 使用@Resource 注入
    @Resource(name = "myBean1")
    InjectBean myBean1;

    // 使用@Autowired 注入
    @Autowired
    InjectBean myBean2;

    @GetMapping("/inject")
    public String inject() {
        System.out.println(myBean1.getId());
        System.out.println(myBean2.getId());
        return "";
    }
}
```

将启动类和控制器写到同一个类中，往这个类中注入两个定义好的bean，分别使用@Resource与@Autowired注解实现注入。运行启动类，在浏览器中输入：http://localhost:8080/

inject，即可以在控制台中看到结果。

可能有细心的朋友会发现，@Autowired 是根据类型注入的，而在上面的例子中，Spring 容器里面有两个 InjectBean 类型的 bean，为什么@Autowired 会拿到相应的 bean 呢？实际上，@Autowired 默认会根据类型来获取 bean，如果存在多个 bean，则会根据属性名来查找，而前面例子中的属性名为 myBean2，因此最终会找到相应的 bean。如果我们在这个例子的基础上，将 myBean2 修改为 myBean3，则在启动时，会得到以下错误信息：

```
    Field myBean3 in org.crazyit.boot.c4.inject.InjectApp required a single bean, but
2 were found:
        - myBean1: defined by method 'myBean1' in class path resource [org/crazyit/
boot/c4/inject/InjectConfig.class]
        - myBean2: defined by method 'myBean2' in class path resource [org/crazyit/
boot/c4/inject/InjectConfig.class]
```

以上错误信息大致的意思是，InjectApp 的 myBean3 属性在容器中找到两个 bean，而 Spring 并不知道应该往控制器注入哪个，所以就报出以上的错误信息。@Resource 注解也会有同样的问题，如果使用@Resource 时根据名称无法查找到 bean，则会根据类型进行注入，如果找到多个相同类型的 bean，则会报出以下的异常信息：

```
    Error creating bean with name 'injectApp': Injection of resource dependencies failed;
    nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException:
    No qualifying bean of type 'org.crazyit.boot.c4.inject.InjectBean'
    available: expected single matching bean but found 2: myBean1,myBean2
```

以上的示例使用了@Resource 与@Autowired 进行依赖注入，注入方式为设值注入。除了这种设值注入外，还可以使用构造注入，向控制器的构造器中注入 bean，如以下代码片断：

```
    InjectBean myBean;

    @Autowired
    public InjectApp(InjectBean myBean2) {
        this.myBean = myBean2;
    }
```

构造注入效果与设值注入类似，在此不再赘述。

注意：@Resource 注解不能修饰构造器，需要使用@Autowired 注解来实现构造注入。

4.1.3 使用 Primary 注解

根据上一节可知，不管是@Resource 还是@Autowired 注解，如果根据类型来注入，而且容器中存在多个同类型的 bean 时，就会抛出异常，因为此时 Spring 并不知道将哪个 bean 注入，针对这个问题，可以使用@Primary 注解，请见代码清单 4-5。

代码清单 4-5：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\primary\PrimaryConfig.java

```
@Configuration
public class PrimaryConfig {

    @Bean
    @Primary
    public PrimaryBean bean1() {
        return new PrimaryBean("1");
    }
```

```
    @Bean
    public PrimaryBean bean2() {
        return new PrimaryBean("2");
    }
}
```

在容器中配置了两个bean，其中bean1使用了@Primary注解，也就是在Spring容器中会存在两个类型均为PrimaryBean的bean，如果在控制器或者其他组件中，根据类型注入PrimaryBean，则会注入bean1，请见代码清单4-6。

代码清单4-6：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\primary\PrimaryApp.java

```
    @Autowired
    PrimaryBean bean;

    @GetMapping("/primary")
    public String primary() {
        System.out.println(bean.getId());
        return "";
    }
```

在代码清单4-6中使用了@Autowired注解进行注入，由于bean1使用了@Primary注解，因此只会注入bean1的实例。如果代码清单4-5没有使用@Primary注解，则会在启动时抛出错误信息，原因在前面小节讲过：根据类型找到两个bean，Spring并不知道应该注入哪个。

运行PrimaryApp.java，使用浏览器访问：http://localhost:8080/primary，则可以看到控制台输出的结果。

4.1.4 Scope注解

在配置bean时，可以指定bean的作用域（scope），一般的bean可以被配置为单态（singleton）或者非单态（prototype）的。配置为singleton的话，Spring的bean工厂都会只返回同一个bean实例，而配置为prototype，则每次都会创建一个新的实例。代码清单4-7定义了两个有不同作用域的bean。

代码清单4-7：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\scope\ScopeConfig.java

```
@Configuration
public class ScopeConfig {

    @Bean
    @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public ScopeBean bean1() {
        return new ScopeBean();
    }

    @Bean
    @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
    public ScopeBean bean2() {
        return new ScopeBean();
    }
}
```

在ScopeConfig类里面定义了两个bean，分别是prototype与singleton。在控制器中使用ApplicationContext来获取这两个bean，并且将它们输出到控制台，请见代码清单4-8。

代码清单 4-8：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\scope\ScopeApp.java

```
    @Autowired
    ApplicationContext ctx;

    @GetMapping("/hello")
    public String hello() {
        System.out.println("非单态的bean: " + ctx.getBean("bean1") + ", 单态的bean: "
                + ctx.getBean("bean2"));
        return "hello";
    }
```

启动服务器，在浏览器中访问：http://localhost:8080/hello，刷新多次查看结果，可以看到非单态的 bean 每次都返回一个新的实例。

单态与非单态的配置较为简单，但是有一点需要注意，如果在一个单态的 bean 里面注入一个非单态的 bean，则这个单态的 bean 所维护的非单态 bean 实例，将不会被刷新。举个简单的例子，Spring MVC 的控制器是单态的，如果往控制器里面注入一个非单态的 bean，例如以下代码片断：

```
@Controller
public class TestController {

    // 注入一个非单态的bean
    @Autowired
    private ScopeBean bean1;

    @GetMapping("/hello2")
    public String hello() {
        System.out.println(bean1);
        return "hello";
    }
}
```

在以上代码片断中，多次调用 hello 方法打印的 ScopeBean 都会是同一个实例，因为控制器在初始化时，就已经被注入了一个 bean，而且一直维护着同一个实例。Spring 的官方文档不建议使用这种方式进行依赖注入，建议使用 Spring 的“方法注入”来解决 bean 实例无法刷新的问题。

@Scope 注解的 value 属性，除了可以取值为“singleton”和“prototype”外，还可以设置为“request”“session”“application”和“websocket”，这些值都需要在 Web 环境中使用，在此不展开讨论。

4.1.5 方法注入

如前面小节所讲的，在一个单态的 bean 中注入一个非单态的 bean，会导致非单态 bean 的实例无法刷新的问题，解决这个问题有多种方法，以下提供两种简单的解决方法：第一，在需要注入的一方（单态的 bean），直接使用 ApplicationContext，每次调用非单态的 bean，都由 Spring 容器返回；第二，使用 Spring 的方法注入。第一种方法，请见代码清单 4-9。

代码清单 4-9：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\mi\MiController.java

```
@RestController
public class MiController {
```

```
    @Autowired
    private ApplicationContext ctx;

    private MiBeanA getMiBean() {
        return (MiBeanA)ctx.getBean("bean1");
    }
}
```

代码中的 MiController 是一个单态的 bean，在它的里面需要使用一个非单态的 bean，类型为 MiBeanA。在上面的代码清单中，并没有直接注入 MiBeanA 的实例，而是注入了 ApplicationContext，当需要使用 MiBeanA 时，再通过 ApplicationContext 来返回。这种方式会导致我们的代码与 ApplicationContext 耦合，并不推荐使用这种方式。

第二种方法，可使用 Lookup Method Injection（方法注入）来解决。在使用实例的一端，添加一个获取非单态 bean 的抽象方法，请见代码清单 4-10。

代码清单 4-10：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\mi\MiController.java

```
@RestController
public abstract class MiController {

    @GetMapping("/mi")
    public String mi() {
        System.out.println(createMiBean());
        return "";
    }

    @Lookup("bean1")
    public abstract MiBeanA createMiBean();
}
```

在该代码清单中使用了@Lookup 注解来修饰一个抽象方法，这个方法会返回 MiBeanA 的实例。运行启动类（本例为 org.crazyit.boot.c4.mi.MiApp），在浏览器中访问：http://localhost:8080/mi，多次刷新，可以看到 MiController 每次都会得到不同的 MiBeanA 实例。实际上，Spring 容器会使用 CGLIB 库，帮我们动态生成一个 MiController 的子类，并且会实现 createMiBean 方法。在实际应用中，一个单态的 bean 使用一个非单态的 bean，这种情况并不多见，如果遇到可以考虑以上介绍的两种方法。

4.1.6 AOP 注解

AOP 是 Spring 的重要功能之一，很多时候，AOP 会被应用在数据库事务、业务日志记录、权限控制等功能中。在 Spring Boot 项目中使用 AspectJ 的注解来实现 AOP 功能，需要为 pom.xml 加入以下依赖：

```
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
        </dependency>
```

新建业务类与代理类，请见代码清单 4-11。

代码清单 4-11：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\aop\SaleServiceImpl.java
codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\aop\ProxyService.java

```
@Component
public class SaleServiceImpl {
```

```
    public void saleService() {
        System.out.println("要代理的销售业务方法");
    }
}

@Aspect
@Component
public class ProxyService {

    @Before("execution(* org.crazyit.boot.c4.aop.*ServiceImpl.*(..))")
    public void before() {
        System.out.println("业务方法调用前执行");
    }

    @After("execution(* org.crazyit.boot.c4.aop.*ServiceImpl.*(..))")
    public void after() {
        System.out.println("业务方法调用后执行");
    }
}
```

在代码清单 4-11 中，要注意代理类，其使用@Aspect 注解进行修饰，使用@Before 与@After 来设置通知（Advice），并为其配置了相应的切点（Pointcut）。代码清单 4-11，简单来说，就是在业务方法调用前执行 before 方法，在业务方法调用后执行 after 方法。新建 Spring Boot 的启动类与控制器，本例的启动类与控制器为同一个类，请见代码清单 4-12。

代码清单 4-12：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\aop\ProxyApp.java

```
@SpringBootApplication
@RestController
public class ProxyApp {

    public static void main(String[] args) {
        new SpringApplicationBuilder(ProxyApp.class).properties(
                "spring.aop.proxy-target-class=true").run(args);
    }

    @Autowired
    SaleServiceImpl saleService;

    @GetMapping("/sale")
    public String saleService() {
        saleService.saleService();
        System.out.println("SaleServiceImpl 的class: " + saleService.getClass());
        return "";
    }
}
```

执行 ProxyApp 类的 main 方法后，打开浏览器访问以下地址：http://localhost:8080/sale，控制台输出如下：

```
业务方法调用前执行
要代理的销售业务方法
业务方法调用后执行
SaleServiceImpl 的class: class org.crazyit.boot.c4.aop.
SaleServiceImpl$$EnhancerBySpringCGLIB$$a6f12b8f
```

根据结果可知，我们的业务方法已经被代理，因此在方法调用前和调用后，都会执行通知的方法，输出的代理类为经过 CGLIB 处理的类。

在 Spring Boot 2.0 中，默认情况下，不管是代理接口还是类，都使用 CGLIB 代理。我们可以将 spring.aop.proxy-target-class 配置为 false，这样在代理接口时，会使用 JDK 动态代理。

在启动类中将 spring.aop.proxy-target-class 设置为 false 后，在浏览器中访问：http://localhost:8080/mer，则可以在控制台中看到，生成的代理类类似于 com.sun.proxy.$Proxy61。Spring AOP 功能强大，这里所述仅是其冰山一角，限于篇幅，更深入的 AOP 功能不再赘述。

4.1.7 ComponentScan 注解

ComponentScan 注解主要用于检测使用@Component 修饰的组件，并把它们注册到 Spring 容器中。除了直接使用@Component 修饰的组件外，还有间接使用@Component 的组件（如@Service、@Repository、@Controller，或自定义注解），都会被检测到。在使用 Spring Boot 时，我们一般很少直接接触 ComponentScan 注解，但实际上每个应用都使用它，它只是被放到@SpringBootApplication 里面了。在使用 ComponentScan 注解时，要注意过滤器，例如，代码清单 4-13 即使用了 ComponentScan 的过滤器。

代码清单 4-13：codes\04\4.1\spring-ants\src\main\java\org\crazyit\boot\c4\scan\ScanApp.java

```
@SpringBootConfiguration
@EnableAutoConfiguration
@RestController
@ComponentScan(basePackages = "org.crazyit.boot.c4.scan", excludeFilters = {
        @Filter(type = FilterType.ASSIGNABLE_TYPE, classes = MyComponent.class)
})
public class ScanApp {

    public static void main(String[] args) {
        SpringApplication.run(ScanApp.class, args);
    }

    @Autowired
    private ApplicationContext ctx;

    /**
     * 调用将会报错：NoSuchBeanDefinitionException: No bean named 'myComponent' available
     * @return
     */
    @GetMapping("/scan")
    public String scan() {
        System.out.println(ctx.getBean("myComponent"));
        return "";
    }
}
```

ScanApp 类使用了@ComponentScan 修饰，设置了 excludeFilters，因此在扫描组件时，将会过滤掉 MyComponent 类，从而容器在启动后，再向 Spring 容器请求“myComponent”的 bean 时，将会抛出异常。启动服务器（ScanApp），在浏览器中访问：http://localhost:8080/scan，可以看到控制台输出的异常信息。

对于 ComponentScan 注解的过滤器，可以指定多个类型，甚至可以使用自定义的过滤器，但由于在实际环境中应用不多，因此不再赘述，读者在见到 ComponentScan 时，大概知道其

表示的意思即可。

4.2 高级 Spring 注解

前面讲解了 Spring 的一些基本注解，在一般情况下，这些注解已经够用了。这一节，我们再深入学习一些 Spring 的高级注解，这部分内容与一些基础的注解也有关联。

4.2.1 限定注解

在前面章节我们讲过一个 Primary 注解，当存在多个同类型的 bean 时，可以指定优先注入的 bean。如果想对 bean 的注入选择做进一步的控制，则可以使用限定注解。限定注解可以与特定的参数关联起来，缩小类型匹配的范围，最后选择一个符合条件的 bean 来注入。代码清单 4-14 使用了限定注解。

代码清单 4-14：codes\04\4.2\ants-adv\src\main\java\org\crazyit\boot\c4\qlf\QuaConfig.java
codes\04\4.2\ants-adv\src\main\java\org\crazyit\boot\c4\qlf\QuaApp.java

```
@Configuration
public class QuaConfig {

    @Bean
    public QuaBean quaBeanA() {
        return new QuaBean("a");
    }

    @Bean
    public QuaBean quaBeanB() {
        return new QuaBean("b");
    }
}

@SpringBootApplication
@RestController
public class QuaApp {

    public static void main(String[] args) {
        SpringApplication.run(QuaApp.class, args);
    }

    @Autowired
    @Qualifier("quaBeanA")
    QuaBean bean;

    @GetMapping("/qua")
    public String testGetBean() {
        System.out.println(bean.getName());
        return "";
    }
}
```

在配置类中，定义了两个同为 QuaBean 类型的 bean，分别设置了不同的 name 属性。在启动类（同时也是控制器）中，使用@Autowired 注入一个 QuaBean 类型的 bean，并且使用@Qualifier 指定注入名称为 quaBeanA 的 bean。运行 QuaApp 后，访问：http://localhost:8080/qua，控制台会输出 QuaBean 的 name 属性。

4.2.2 自定义限定注解

前面我们使用了限定注解，当存在两个相同类型的注解时，可以根据 bean 的名称来指定注入的 bean。如果需要根据特定的属性来指定注入的 bean，则可以自定义限定注解。新建一个自定义注解，并且在配置 bean 时，使用这个注解，请见代码清单 4-15。

代码清单 4-15：codes\04\4.2\ants-adv\src\main\java\org\crazyit\boot\c4\ct\AnimalQualifier.java
codes\04\4.2\ants-adv\src\main\java\org\crazyit\boot\c4\ct\CustomConfig.java

```
@Target({ ElementType.FIELD, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface AnimalQualifier {

    String type();
}

@Configuration
public class CustomConfig {

    @Bean
    @AnimalQualifier(type = "person")
    public AnimalBean personBean() {
        return new AnimalBean("angus"); // 动物类型为 person，名称为 angus
    }

    @Bean
    @AnimalQualifier(type = "cat")
    public AnimalBean catBean() {
        return new AnimalBean("tom"); // 动物类型为 cat，名称为 tom
    }
}
```

在该代码清单中定义了一个 AnimalQualifier 的注解，该注解使用了@Qualifier 修饰，并且需要设置 type 属性。在配置 bean 时，需要为相应的 bean 设置不同的类型。对于代码清单 4-15，可以这样理解，Spring 容器提供了不同类型的动物 bean，并使用@AnimalQualifier 为它们进行了标识。在使用这些 bean 时，同样使用@AnimalQualifier 并指定 type 属性，就可以拿回相应的 bean。代码清单 4-16 在控制器中获取 bean。

代码清单 4-16：codes\04\4.2\ants-adv\src\main\java\org\crazyit\boot\c4\ct\CustomApp.java

```
@SpringBootApplication
@RestController
public class CustomApp {

    public static void main(String[] args) {
        SpringApplication.run(CustomApp.class, args);
    }

    @Autowired
    @AnimalQualifier(type = "person")
    private AnimalBean person;

    @GetMapping("/cus")
    public String getAnm() {
        System.out.println(person.getName());
```


仅供非商业用途或交流学习使用

```
        return "";
    }
}
```

在控制器中注入 bean 时，额外使用了@AnimalQualifier 注解，指定 type 为“person”。运行 CustomApp 后，访问：http://localhost:8080/cus，控制台输出：angus，这说明控制器已经得到期望的 bean。

4.2.3 自定义 bean 的生命周期

前面介绍了，Scope 注解主要用于配置 bean 在容器中的生命周期，常被配置为 singleton（单态）与 prototype（非单态）的，在 Web 环境下，还可以配置为 request、session 等值，表示 Spring 容器会为一次请求或一个会话分配一个 bean 的实例。如果对 bean 的生命周期有特殊的需求，则可以考虑使用自定义的 Scope，自己写代码来定义 bean 的生命周期。

举一个特别的例子，假设有这样一个需求，一个 bean 被使用 4 次后，就需要获取新的 bean 实例，针对这样的需求，可编写一个自定义的 Scope，请见代码清单 4-17。

代码清单 4-17：codes\04\4.2\ants-adv\src\main\java\org\crazyit\boot\c4\scope\MyScope.java

```
public class MyScope implements Scope {

    // 记录 bean 的使用次数
    private Map<String, Integer> beanCounts = new HashMap<String, Integer>();

    // 保存实例
    private Map<String, Object> beans = new HashMap<String, Object>();

    @Override
    public Object get(String name, ObjectFactory<?> objectFactory) {
        if(beanCounts.get(name) == null) {
            beanCounts.put(name, 0);
        }
        // 第一次使用，放到实例的 Map 中
        Integer beanCount = beanCounts.get(name);
        if(beanCount == 0) {
            Object newObject = objectFactory.getObject();
            beans.put(name, newObject);
        }
        Object bean = beans.get(name);
        // 计数器加 1
        Integer newBeanCount = beanCount + 1;
        // 使用次数超过 4，设置为 0
        if(newBeanCount >= 4) {
            newBeanCount = 0;
        }
        // 设置新的次数
        beanCounts.put(name, newBeanCount);
        //  返回实例
        return bean;
    }

    ...省略其他的实现方法
}
```


仅供非商业用途或交流学习使用

代码清单 4-17 实现了 Scope 接口，添加了两个 Map，一个 Map 用于保存 bean 被调用的次数，另一个 Map 则用于保存 bean 实例，两个 Map 的 key 均为 bean 的名称。代码清单 4-17 的逻辑较为简单，即 bean 每次被调用时，都会让计数器加 1，当一个 bean 的实例调用次数超过 4 次时，计数器会被重新设置为 0，此时会设置一个新的实例放到 bean 实例的 Map 中。

注意：本例并没有实现其他 Scope 接口方法。

接下来，需要将我们自定义的 Scope 注册到 Spring 容器，新建一个配置类，请见代码清单 4-18。

代码清单 4-18：codes\04\4.2\ants-adv\src\main\java\org\crazyit\boot\c4\scope\ScopeConfig.java

```
@Configuration
public class ScopeConfig {

    @Autowired
    BeanFactory factory;

    @PostConstruct
public void customScopeConfigurer() {
    // ScopeConfig 初始化后执行本方法，创建 MyScope 的实例
        CustomScopeConfigurer config = new CustomScopeConfigurer();
        config.addScope("four", new MyScope());
        config.postProcessBeanFactory((ConfigurableListableBeanFactory)factory);
    }

    @Bean
    @Scope(scopeName = "four")
    public ScopeBean beanA() {
        return new ScopeBean();
    }
}
```

ScopeConfig 的 customScopeConfigurer 方法，新建了一个 CustomScopeConfigurer 类，添加了一个名称为“four”的 Scope，这就是我们的自定义 Scope。在 ScopeConfig 中还定义了一个 ScopeBean，该 bean 使用@Scope 进行修饰，指定使用的 Scope 为“four”。

编写控制器，直接使用 ApplicationContext 来获取 ScopeBean 的实例，验证自定义生命周期是否生效。启动类（同控制器）的代码请见代码清单 4-19。

代码清单 4-19：codes\04\4.2\ants-adv\src\main\java\org\crazyit\boot\c4\scope\ScopeApp.java

```
@SpringBootApplication
@RestController
public class ScopeApp {

    public static void main(String[] args) {
        SpringApplication.run(ScopeApp.class, args);
    }

    @Autowired
    ApplicationContext ctx;

    @GetMapping("/scope")
    public String scope() {
        for(int i = 0; i < 6; i++) {
```


仅供非商业用途或交流学习使用

```
            System.out.println(ctx.getBean("beanA"));
        }
        return "";
    }
}
```

启动 Spring Boot 应用后，访问：http://localhost:8080/scope，控制台输出如下：

```
org.crazyit.boot.c4.scope.ScopeBean@4b0578
org.crazyit.boot.c4.scope.ScopeBean@4b0578
org.crazyit.boot.c4.scope.ScopeBean@4b0578
org.crazyit.boot.c4.scope.ScopeBean@4b0578
org.crazyit.boot.c4.scope.ScopeBean@1fa33f4
org.crazyit.boot.c4.scope.ScopeBean@1fa33f4
```

前面 4 次 getBean 调用，都会得到同一个 ScopeBean 实例，在第 5 次调用 getBean 时，会得到一个全新的 ScopeBean 实例。根据结果可知，我们自定义的 Scope 已经生效。在实际应用中，需要自定义 bean 生命周期的情况并不多见，Spring 内置的几个 Scope 基本可以满足我们的需求。

4.3 Spring MVC 的常用注解

前面讲述了 Spring 框架的常用注解，而 Spring Boot 默认集成了 Spring MVC。这一节，我们会讲解一些 Spring MVC 的常用注解，在本书中会经常使用这些注解，它们可完成大部分的 Web 功能。

4.3.1 Controller 注解

Spring MVC 提供了 Controller 等注解，用于修饰 Java 类，被修饰的类会充当 MVC 中的控制器角色，它们不需要继承其他类，也不需要实现接口。Controller 注解使用了@Component 修饰，换言之，使用了 Controller 修饰的类，会被@ComponentScan 检测，并且会作为 Spring 的 bean 被放到容器中。代码清单 4-20 即使用了 Controller 等注解。

代码清单 4-20：codes\04\4.3\mvc-ants\src\main\java\org\crazyit\boot\c4\ctl\MyController.java

```
@Controller
public class MyController {

    @RequestMapping(value = "/hello")
    @ResponseBody
    public String hello() {
        return "hello";
    }
}
```

访问应用的“/hello”路径，将会得到“hello”字符串。除了 Controller 注解外，还有一个 RestController 注解，这是一个组合注解，它由@Controller 与@ResponseBody 组成。使用了@RestController 修饰的类，默认情况下其每个控制器方法都等于使用了@ResponseBody 修饰，方法的返回值会被写到响应体中。以下代码等同于代码清单 4-20。

```
@RestController
public class MyRestController {
```


仅供非商业用途或交流学习使用

```
    @RequestMapping(value = "/rhello")
    public String rhello() {
        return "rest hello";
    }
}
```

启动服务器（codes\04\4.3\mvc-ants\src\main\java\org\crazyit\boot\c4\ctl\ControllerApp.java），在浏览器中访问：http://localhost:8080/hello 与 http://localhost:8080/rhello，可看到具体效果。

4.3.2 RequestMapping 注解

RequestMapping 注解可修饰类或方法，主要用于映射请求与处理方法。当使用它来修饰一个类并设置了 URL 时，可以理解为，为各个请求设置了 URL 前缀，请见代码清单 4-21。

代码清单 4-21：codes\04\4.3\mvc-ants\src\main\java\org\crazyit\boot\c4\rm\RmApp.java

```
@SpringBootApplication
@RestController
@RequestMapping("/oa")
public class RmApp {

    public static void main(String[] args) {
        SpringApplication.run(RmApp.class, args);
    }

    @RequestMapping("/hello")
    public String hello() {
        return "hello";
    }
}
```

在该代码清单中，Spring Boot 的启动类与控制器为同一个。在类一级使用 @RequestMapping 配置访问前缀为“/oa”，如果要请求“hello”方法，则访问路径：http://localhost:8080/oa/hello。RequestMapping 注解主要有以下属性。

- path 与 value：两个属性互为别名，用于配置映射的 URI。在配置 RequestMapping 时，如果不设置属性名（如 RequestMapping(“hello”)），则等同于配置了 path 属性。
- method：映射的 HTTP 方法，例如 GET、POST、PUT、DELETE 等。
- params：为映射的请求配置参数标识，例如配置为“params = 'action=save'”，则要通过“http://localhost:port/hello?action=save”访问该映射的方法。在表单中，可以为表单提交按钮设置 name 属性，例如<input type="submit" name="add"/>，则表单会被提交到 params 值为 add 的方法。
- consumes：配置请求的数据类型，可配置为 XML 或 JSON 等数据格式。
- produces：配置响应的数据类型，例如配置为“application/json”，则表示返回 JSON 数据。

为了让我们减少配置，Spring MVC 还提供了几个注解，这些注解默认配置了 @ReqeustMapping 的 method 属性。例如@GetMapping 注解，其效果等同于@RequestMapping (method = “RequestMethod.GET”)；还有@PostMapping、@PutMapping、@DeleteMapping、@PatchMapping，这几个注解默认都设置了不同的 method 属性。

仅供非商业用途或交流学习使用

4.3.3 PathVariable 注解

@PathVariable 主要用来修饰方法参数，表示该方法参数是请求 URI 的变量。请求 URI 中的变量，会自动转化为方法的参数类型，请见代码清单 4-22。

代码清单 4-22：codes\04\4.3\mvc-ants\src\main\java\org\crazyit\boot\c4\pv\PvApp.java

```
@RequestMapping(value = "/hello/{name}")
@ResponseBody
public String hello(@PathVariable String name) {
    return "hello, " + name;
}
```

以上代码中的方法参数“name”使用了@PathVariable 修饰，这样，在处理请求 URI 时，会自动将地址中的{name}变量转化为方法的 name 参数。默认情况下，URI 中的变量名称为 name，因此该变量会与方法中名称为 name 的参数绑定，当然，我们也可以显式指定，请见以下代码：

```
@RequestMapping(value = "/test/{myName}")
@ResponseBody
public String test(@PathVariable("myName") String abc) {
    return "test, " + abc;
}
```

在以上代码片断中，为@PathVariable 配置了属性值，这个属性值显式绑定了 “abc”参数与 “myName”变量。

4.3.4 MatrixVariable 注解

前面的@PathVariable 注解主要用来获取单一的 URI 参数，如果想通过 URI 来传输一些复杂的参数，则可以考虑使用@MatrixVariable 注解。使用时需要遵守一定的规范，参数的名称与值使用 key-value 的形式，多个参数间使用分号（;）隔开，如果一个参数拥有多个值，则值与值之间使用逗号隔开。这种 URI 的参数表现形式，称为矩阵变量，请见以下的 URI：

```
/car/{id};color=red;year=2012
```

该 URI 传入了 id 参数，并且传入 color 与 year 两个参数，值分别为 red 与 2012。对于上面的 URI，在控制器中如何获取呢？首先，需要开启 Spring MVC 对矩阵变量的支持。在 Spring Boot 中，新建配置类，请见代码清单 4-23。

代码清单 4-23：codes\04\4.3\mvc-ants\src\main\java\org\crazyit\boot\c4\mv\MyWebConfig.java

```
@Configuration
public class MyWebConfig extends WebMvcConfigurationSupport {

    @Override
    public RequestMappingHandlerMapping requestMappingHandlerMapping() {
        final RequestMappingHandlerMapping requestMappingHandlerMapping = super
                .requestMappingHandlerMapping();
        requestMappingHandlerMapping.setRemoveSemicolonContent(false);
        return requestMappingHandlerMapping;
    }
}
```

在代码清单 4-23 中，将容器中 RequestMappingHandlerMapping 的 removeSemicolonContent 属性设置为 false，即可开启对矩阵变量的支持。接下来，新建启动类与控制器，在方法中获

仅供非商业用途或交流学习使用

取矩阵变量，请见代码清单 4-24。

代码清单 4-24：codes\04\4.3\mvc-ants\src\main\java\org\crazyit\boot\c4\mv\MvApp.java

```
@SpringBootApplication
@Controller
public class MvApp {

    public static void main(String[] args) {
        SpringApplication.run(MvApp.class, args);
    }

    @GetMapping(value = "/car/{id}")
    @ResponseBody
    public String pathA(@PathVariable Integer id,
            @MatrixVariable String color,
            @MatrixVariable String year) {
        System.out.println(id);
        System.out.println(color);
        System.out.println(year);
        return "test";
    }
}
```

运行代码清单 4-24，在浏览器中访问：http://localhost:8080/car/100;color=red;year=2012，则可以看到控制台会输入相应的变量。矩阵变量还有其他的表现形式，但在日常开发中并不常用，本章就不深入讨论了。

4.3.5 RequestParam 注解

如果要获取请求中的参数，则可以使用@RequestParam 注解。当一个请求参数被放到请求体中时，就可以使用这种方式获取它，最常见的就是表单的提交。新建一个控制器方法，用于获取请求参数，请见代码清单 4-25。

代码清单 4-25：codes\04\4.3\mvc-ants\src\main\java\org\crazyit\boot\c4\rp\RpApp.java

```
@RequestMapping(value = "/login")
@ResponseBody
public String login(@RequestParam String userName) {
    System.out.println(userName);
    return "success";
}
```

为了简单起见，本章不对表现层进行配置。新建一个最简单的 HTML 文件（该文件不需要在服务器中运行，可以直接在浏览器打开），向控制器提交表单，请见代码清单 4-26。

代码清单 4-26：codes\04\4.3\mvc-ants\form.html

```
<form method="post" action="http://localhost:8080/login">
    <input type="text" name="userName"/>
    <input type="submit" value="提交"/>
</form>
```

@RequestParam 注解还可以用来修饰 Map 或者 MultiValueMap，MultiValueMap 继承于 Map 接口，我们所得实例为 LinkedMultiValueMap。注意，该 Map 实现并非线程安全的。在 Map 中将会存放所有的请求参数，请见以下代码片断：

仅供非商业用途或交流学习使用

```
    @RequestMapping(value = "/multi")
    @ResponseBody
    public String multi(@RequestParam Map<String, String> values) {
        System.out.println(values.get("myName"));
        System.out.println(values.get("passwd"));
        return "success";
    }
```

4.3.6 文件上传

文件上传是常见的功能，使用 RequestParam 注解就可以在表单中实现文件上传。获取文件内容的控制器方法，请见代码清单 4-27。

代码清单 4-27：codes\04\4.3\mvc-ants\src\main\java\org\crazyit\boot\c4\file\FileApp.java

```
    @RequestMapping("/upload")
    @ResponseBody
    public String upload(@RequestParam("file") MultipartFile file)
            throws IOException {
        byte[] content = file.getBytes();
        System.out.println(content);
        return "success";
    }
```

在控制器中，使用@RequestParam 修饰 MultipartFile 参数，在方法体中直接输出文件的 byte 数组。接下来，编写一个静态的 HTML 页面，实现文件上传。再次强调，前一小节以及本小节的 HTML 页面，都不需要运行在服务器中，直接在浏览器中打开即可。文件上传的 HTML 页面内容，请见代码清单 4-28。

代码清单 4-28：codes\04\4.3\mvc-ants\file.html

```
    <form method="post" action="http://localhost:8080/upload" enctype=
"multipart/form-data">
        <input type="file" name="file"/>
        <input type="submit" value="提交"/>
    </form>
```

直接使用浏览器打开 HTML 页面，选择文件并提交表单后，则可以看到控制台的输出。

4.3.7 RestController 注解

RestController 注解的目的是为了让我们更加简便地使用@Controller 与@ResponseBody 这两个注解，查看@RestController 源码就可以知道，这个注解本身就使用了@Controller 与@ResponseBody 来修饰。ResponseBody 注解可以修饰控制器方法，方法的返回值将会被写到 HTTP 的响应体中，所返回的内容，将不放到模型中，也不会被解释为视图的名称。例如以下代码片断，将会直接返回字符串：

```
@RestController
public class RcApp {

    @RequestMapping("/hello")
    public String hello() {
        return "Hello World";
    }
}
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

除了本节所介绍的注解外，还有 RequestAttribute、SessionAttribute 等注解也较为常用，这些注解，会配合页面、AJAX 等知识点，或在具体的案例中一并讲述。这里我们能掌握 Controller、RequestMapping、参数传递等注解即可，更深入的知识，可在后面章节，配合具体的功能予以学习。

4.4 Spring Boot 的条件注解

Spring Boot 提供了自动配置，一些内置的或者常用的组件，Spring 已经帮我们配置好了，我们在使用这些组件时，加入相应项目的依赖就可以了，并不需要再额外进行配置，从而减少了搭建项目的工作量。在学习自动配置前，我们需要先了解条件注解，自动配置是否执行，取决于条件注解，所以我们先来学习条件注解，然后就可以轻松掌握自动配置。

我们先学习几个最常用的条件注解，最后实现一个简单的条件注解。

4.4.1 类条件注解

当类路径中存在或不存在某个类时，再决定是否执行自动配置。存在该类则使用@ConditionalOnClass，不存在则使用@ConditionalOnMissingClass，它们都可以修饰类或者方法。这两个注解通过配置 value 属性来指定一个类，这个类有可能并不存在于实际的运行环境中，也可以使用 name 属性来指定全限定类名。假设项目中存在 MySQL 的驱动类，则会启用某个配置类，此时可以使用@ConditionalOnClass，请见代码清单 4-29。

代码清单 4-29：codes\04\4.4\cds-ants\src\main\java\org\crazyit\boot\c4\oc\OnClassConfig.java

```
@ConditionalOnClass(name = "com.mysql.jdbc.Driver")
@Configuration
public class OnClassConfig {

    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

当项目的类路径中存在 MySQL 驱动类时，就会创建 MyBean 的实例。我们编写一个启动类来测试以上配置，见代码清单 4-30。

代码清单 4-30：codes\04\4.4\cds-ants\src\main\java\org\crazyit\boot\c4\oc\OnClassApp.java

```
@SpringBootApplication
@RestController
public class OnClassApp {

    public static void main(String[] args) {
        SpringApplication.run(OnClassApp.class, args);
    }

    @Autowired
    ApplicationContext ctx;

    @RequestMapping("/bean")
    public String getBean() {
        Map<String, MyBean> beans = ctx.getBeansOfType(MyBean.class);
        System.out.println(beans.size());
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            return "success";
        }
    }
```

代码清单中的 getBean 方法会到容器中获取 MyBean 类型的 bean，并以 Map 的形式返回，最后输出 Map 的 size。在 pom.xml 文件中，加入 MySQL 依赖，输出的结果为 1，将 MySQL 依赖注释掉，则输出结果为 0。本例的 pom.xml 文件路径为 codes\04\4.4\cds-ants\ pom.xml，在默认情况下会将 MySQL 依赖注释掉。使用@ConditionalOnMissingClass 则有相反的效果，在此不再赘述。

4.4.2 项目依赖问题

Spring Boot 的自动配置项目，很多时候会使用@ConditionalOnClass 注解的 value 来配置类。使用该属性，需要提供具体的类（非类名），也就是说，在自动配置项目的类路径中，**总是存在该类的**。我们的项目在引入自动配置项目时，却不会自动引入这个配置的类，这是由于在 pom.xml 中使用了 optional 属性：

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>6.0.4</version>
            <optional>true</optional>
        </dependency>
```

以上的讲述可能有点抽象，我们举个简单的例子来描述。假设有一个自动配置的项目 B，指定了当运行环境存在项目 A 时，会执行自动配置，项目 C 引用了项目 B，项目 C 是我们实际运行的项目，如果项目 B 在配置项目 A 的依赖时使用了 optional 属性，则默认情况下，项目 C 是不会引入项目 A 的，也就是说，不会执行项目 B 的自动配置，想要执行自动配置，要在项目 C 中引入项目 A。

自动配置的项目 B 引入项目 A 时如果使用 optional 属性，其结果如图 4-1 所示。

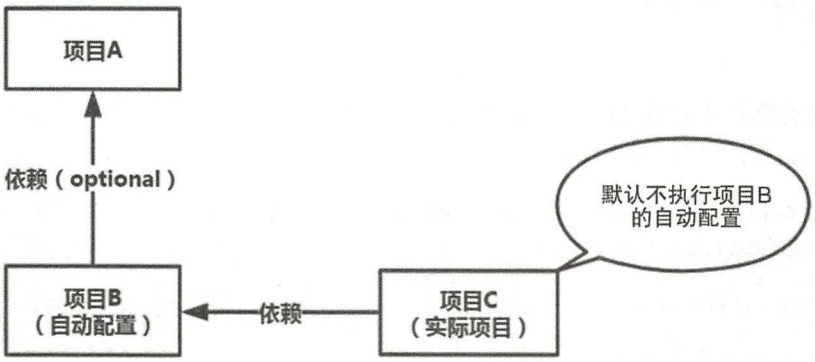


图 4-1　不执行自动配置

自动配置的项目 B 引入项目 A 时如果不使用 optional 属性，其结果如图 4-2 所示。

注意，使用@ConditionalOnClass 的 value 属性时才会存在以上的问题，如果直接使用 name 属性则可避免该问题，因此若使用 name 属性，则可以直接传入全限定类名的字符串，项目 B 则可以不用引入项目 A。上面的依赖关系，并不要求一定要掌握，但是读者一定要清楚两个类条件注解的作用。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

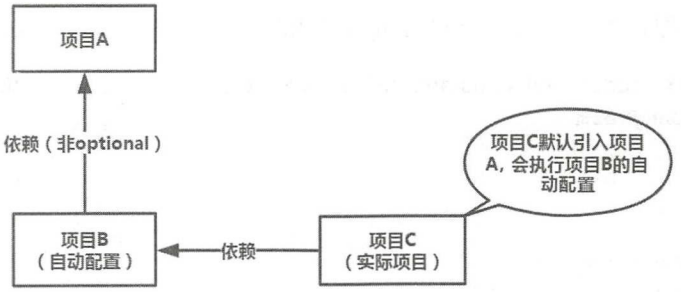


图 4-2 默认会执行自动配置

4.4.3 其他条件注解

除了类条件注解之外，还有其他的条件注解，例如 Bean 条件（Bean Conditions）、属性条件（Property Conditions）、资源条件（Resource Conditions）等，这些条件注解大同小异，在此就不一一描述了。接下来，我们实现一个自己的条件注解。

4.4.4 自定义条件注解

为了让大家更深入地了解条件注解，这一节，我们编写一个最简单的条件注解。新建一个自定义的条件类，请见代码清单 4-31。

代码清单 4-31：codes\04\4.4\cds-ants\src\main\java\org\crazyit\boot\c4\mc\MyCondition.java

```
public class MyCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context,
            AnnotatedTypeMetadata metadata) {
        Environment env = context.getEnvironment();
        String user = env.getProperty("jdbc.user");
        System.out.println("在条件类中获取属性值：" + user);
        // 这个属性值为 crazyit 的时候，条件符合
        if("crazyit".equals(user)) {
            return true;
        }
        return false;
    }
}
```

自定义条件类的 matches 方法会读取环境中的“jdbc.user”属性，如果属性值为“crazyit”，则返回 true，否则返回 false，该属性直接配置在 application.properties 中，当 matches 方法返回 true 时，表示条件成立。接下来，新建一个自定义注解，请见代码清单 4-32。

代码清单 4-32：codes\04\4.4\cds-ants\src\main\java\org\crazyit\boot\c4\mc\ConditionalOnMyTerm.java

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(MyCondition.class)
public @interface ConditionalOnMyTerm {

}
```

注意粗体部分代码，这里使用了@Conditional 注解，传入我们的自定义条件类。按照与前

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

面同样的方法，在配置类中使用这个 ConditionalOnMyTerm 注解，请见代码清单 4-33。

代码清单 4-33：codes\04\4.4\cds-ants\src\main\java\org\crazyit\boot\c4\mc\MyConfig.java

```
@ConditionalOnMyTerm
@Configuration
public class MyConfig {

    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

当条件成立时，创建启用这个配置类，编写控制器，读取容器中的这个 bean，请见代码清单 4-34。

代码清单 4-34：codes\04\4.4\cds-ants\src\main\java\org\crazyit\boot\c4\mc\MyTermApp.java

```
    @Autowired
    ApplicationContext ctx;

    @RequestMapping("/bean")
    public String getBean() {
        Map<String, MyBean> beans = ctx.getBeansOfType(MyBean.class);
        System.out.println(beans.size());
        return "success";
    }
```

读者在测试时，可直接启动 MyTermApp 并访问：http://localhost:8080/bean，当 application.properties 中的 jdbc.user 属性值为“crazyit”时，控制器方法会输出 1，修改为其他值，则会输出 0。

在实际应用中，我们基本上不用编写条件注解，这里编写这个自定义条件注解的目的是为了让大家对 Spring Boot 的条件注解有更深入的了解。

4.4.5 Spring Boot 的自动配置

当我们的项目需要使用一个开源框架的时候，会先将相关依赖的 jar 包引入项目中，然后再对其进行配置。以 Hibernate 为例，要将这个框架整合进来，需先加入 Maven 依赖，再配置数据源、SessionFactory 等组件。Spring Boot 的自动配置，就是为了减少这些组件的配置工作量，对于很多的框架，我们只需要引入依赖，就可以实现“拿来即用”，繁杂的配置工作，Spring 都已经帮我们完成了。

Spring 容器在启动时，会加载 META-INF 下的 spring.factories 文件，并读取 EnableAutoConfiguration 键，之后会解析一堆需要配置的类，这些类都使用了@Configuration 修饰，Spring 容器是否加载这些配置，由这些类里面的条件注解决定。

例如 Spring Boot 的自动配置项目（spring-boot-autoconfigure），其下面的 spring.factories 文件，设置了 HibernateJpaAutoConfiguration，这个类使用了@ConditionalOnClass 条件注解。当类路径中存在 EntityManager 等类时，就会激活这些配置，Hibernate 相关的 bean 就会被创建，从而运行的应用就可以直接使用 Hibernate。

Spring Boot 提供的自动配置，帮我们集成了许多框架，这些框架基本上与各个 starter 对应，Spring Boot 都已经帮我们配置好了这些框架，我们只要引入相应的 starter 依赖就可以马

仅供非商业用途或交流学习使用

上使用了，例如我们的项目要使用 Thymeleaf，则只需要在项目中加入“spring-boot-starter-thymeleaf”依赖就可以了。在本书的后面章节，在整合不同技术框架时，会接触到这些 starter 模块。我们知道了自动配置的原理，再去使用就会更加得心应手。

4.5 本章小结

本章像一个大杂烩，讲解了常用的 Spring 注解，包括 Spring 容器的核心注解、Spring MVC 的注解、Spring Boot 的注解。通过本章的学习，我们基本上可以掌握本书的大部分注解，为学习后面的课程打下基础。在本章的最后，讲解了 Spring Boot 的条件注解与自动配置。了解了自动配置的机制，我们才能更加深入地认识大量的可以拿来即用的组件，才会熟悉它们的配置与创建过程。

仅供非商业用途或交流学习使用

CHAPTER

5

第5章 初试 Thymeleaf

本章要点

- 什么是 Thymeleaf
- 试用 Thymeleaf 的 API
- 在 Web 项目中使用 Thymeleaf
- Spring Boot 整合 Thymeleaf

仅供非商业用途或交流学习使用

在前面章节我们学习了 Spring Boot 的基础知识，我们现在基本上可以使用 Spring Boot 来开发 Web 项目了。而一个 Web 项目，最重要的就是其表现层技术，笔者以前接触最多的是 JSP，那么本章将介绍一个新的模板引擎 Thymeleaf。

5.1 Thymeleaf

这一节我们来大致介绍一下 Thymeleaf 框架，看看它可以为我们带来什么。

5.1.1 什么是 Thymeleaf

Thymeleaf 是一个表现层的模板引擎，一般被使用在 Web 环境中，它可以处理 HTML、XML、JS 等文档，简单来说，它可以将 JSP 作为 Java Web 应用的表现层，有能力展示与处理数据。笔者写作本书之时，Thymeleaf 的最新版本为 3.0.9，本书将以该版本为基础来讲述。

Thymeleaf 可以让表现层的界面节点与程序逻辑被共享，这样的设计，可以让界面设计人员、业务人员与技术人员都参与到项目开发中。下面是一段含有 Thymeleaf 语句的 HTML：

```
<input type="text" name="userName" value="用户名" th:value="${user.name}" />
```

以上这段 HTML 含有 Thymeleaf 的逻辑，大部分的 Thymeleaf 表达式都直接被设置到 HTML 节点中，作为 HTML 节点的一个属性存在。这样，同一个模板文件，既可以使用浏览器直接打开，也可以放到服务器中用来显示数据，并且样式之间基本上不会存在差异，因此界面设计人员与程序设计人员可以使用同一个模板文件，来查看静态与动态数据的效果。

5.1.2 支持的模板类型

Thymeleaf 支持多种模板类型：HTML、XML、JavaScript、CSS 及普通文本等，其默认提供了 6 种模板处理模式：HTML、XML、TEXT、JAVASCRIPT、CSS 与 RAW，前面五种模式，看名字就知道会处理哪种文件，最后一个 RAW 模式，表示不会对模板进行处理。下面，我们分别来介绍 Thymeleaf 对几个模板文件的处理。

5.1.3 HTML 模板

HTML 模式可以处理任何 HTML 输入，包括 HTML5、HTML4 与 XHTML，并且处理时不会对 HTML 格式进行严格的验证，所以我们要尽可能保证模板结构的准确。例如可以成功处理以下的 HTML 语句：

```
<input type="text" th:value="angus">
```

该 HTML 语句结尾缺少了“/”，对于这种情况，HTML 模式并不会抛出异常。而对于以下的 HTML 语句，则会抛出异常：

```
<input type="text" th:value="angus"
```

这段 HTML 语句缺少了“/>”，Thymeleaf 在处理时会抛出以下异常：

```
org.attoparser.ParseException: (Line = 1, Column = 1) Incomplete structure: "<input type='' th:value='angus'"
```

由此可知，Thymeleaf 在处理 HTML 时，并不会进行严格的验证，但对于一些明显的格式错误，会抛出异常。

仅供非商业用途或交流学习使用

5.1.4 XML 模板

相对于 HTML，Thymeleaf 对 XML 模板的处理就严格一些，当 XML 模板中存在没有关闭的节点，节点中的属性没有引号时，Thymeleaf 都会抛出异常，但其对 XML 不会进行实质性的验证，例如一份 XML 不符合 DTD 或者 Schema 规范，就不会抛出异常。简单来说，Thymeleaf 在处理 XML 时，只会对文档进行形式上的检查，而不会进行实质性的检查。

Thymeleaf 在处理以下的两段 XML 时，都会抛出异常：

```
<bean id=""><bean>
<bean id=></bean>
```

第一段 XML 的 bean 节点没有关闭，执行它时将会抛出异常，异常信息为“Malformed markup: element "bean" is never closed”。第二段 XML 的 bean 节点中的 id 属性没有使用引号，同样会抛出异常：“Malformed markup: Value for attribute "id" must be surrounded by quotes”。

除了 HTML 与 XML 模板外，还能处理 JavaScript、CSS 等模板，在此不展开讨论。

5.2 初试 Thymeleaf API

前面我们大概了解了 Thymeleaf 框架，大家知道它扮演的角色（表现层）、能处理什么类型的模板就可以了。这一节，我们来试用 Thymeleaf 的 API，以进一步了解该框架。新建一个普通的 Maven 项目，为项目加入 Thymeleaf 的依赖：

```
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf</artifactId>
    <version>3.0.9.RELEASE</version>
</dependency>
```

5.2.1 处理 HTML

使用 Thymeleaf 等模板引擎开发应用界面，需要按一定的语法规则将逻辑写入 HTML 页面中，然后再把这些“加工”过的 HTML 字符串交给 Thymeleaf 去处理，Thymeleaf 的处理器会解析它们，最后输出真正的、被渲染过的 HTML。

下面，我们编写一段含有 Thymeleaf 语句的 HTML 代码，交给 Thymeleaf 的 API 去处理，请见代码清单 5-1。

代码清单 5-1：codes\05\5.2\first-th\src\main\java\org\crazyit\boot\c5\HtmlTest.java

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;

public class HtmlTest {

    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        Context ctx = new Context();
        String result = templateEngine.process("<input type='' th:value='angus'/>",ctx);
        System.out.println(result);
    }
}
```

代码清单 5-1 中的粗体字代码是含有 Thymeleaf 逻辑的 HTML 代码，将其交给 TemplateEngine 进行处理，运行后输出如下：

```
<input type='' value='angus'/>
```

根据输出结果可知，TemplateEngine 类会将这些含有逻辑的代码转换成纯 HTML 输出，这就是模板引擎的作用。我们只要按照模板引擎的语法规则来编写 HTML 页面，再将页面交给模板引擎去处理，最终就会输出纯 HTML 字符串。

5.2.2 更换模板解析器

在默认情况下，模板处理模式为 HTML。我们可以设置不同的解析器，让 Thymeleaf 帮我们处理其他类型的文本，例如 XML。代码清单 5-2 为一个处理含有 Thymeleaf 逻辑的 XML 语句的示例。

代码清单 5-2：codes\05\5.2\first-th\src\main\java\org\crazyit\boot\c5\XmlTest.java

```
public class XmlTest {

    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        // 新建模板解析器
        StringTemplateResolver resolver = new StringTemplateResolver();
        // 设置模板模式为 XML
        resolver.setTemplateMode(TemplateMode.XML);
        // 将解析器设置到引擎实例中
        templateEngine.setTemplateResolver(resolver);
        Context ctx = new Context();
        // 处理一段含有 Thymeleaf 逻辑的 XML 语句
        String result = templateEngine.process(
                "<bean id=\"myBean\" th:attr='class=org.crazyi.Test'></bean>",
                ctx);
        System.out.println(result);
    }
}
```

在代码清单 5-2 中，新建了一个类型为 StringTemplateResolver 的模板解析器，处理的模板类型为 XML。最后将一段 XML 交给模板引擎去处理，这一段 XML 中同样含有 Thymeleaf 的逻辑（使用了 th:attri 属性）。运行代码清单 5-2，输出结果如下：

```
<bean id="myBean" class="org.crazyi.Test"></bean>
```

根据结果可知，Thymeleaf 帮我们处理了模板语句，最终输出了 XML。除了 HTML 和 XML 外，Thymeleaf 也可以帮我们处理 JavaScript、CSS，甚至是普通的文本。

5.2.3 处理资源文件

在实际开发中，我们不可能将一大串 HTML 字符串交给模板引擎去处理，一般都是告诉它去找某个文件。可以为模板引擎设置资源解析器，让它去找文件来处理。假设在项目的“src/main/resources”下有一个 HTML 文件，该文件中含有 Themeleaf 的逻辑，则我们可以使用 API 去处理它，详细请见代码清单 5-3。

代码清单 5-3：codes\05\5.2\first-th\src\main\java\org\crazyit\boot\c5\ClassLoaderTest.java

```
public class ClassLoaderTest {
```

```
    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        // 创建解析器
        ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
        // 设置到模板引擎中
        templateEngine.setTemplateResolver(resolver);
        // 处理 classpath 下的 index.html 文件
        String result = templateEngine.process("index.html", new Context());
        System.out.println(result);
    }
}
```

在代码清单 5-3 中，使用了 ClassLoaderTemplateResolver 这个解析器，它在处理文件时，直接将类路径下的 index.html 交给模板引擎处理，index.html 的内容如下：

```
<input type="text" th:value="angus"/>
```

到了这里，大家可能已经明白了，如果把上面的这段代码移到我们的 Web 项目中，就可以直接处理 HTML 页面，并返回给浏览器。在后面的章节，我们会讲解如何在 Web 项目中使用 Thymeleaf，那时你就会知道，我们的页面会经过什么样的处理过程。

5.2.4 变量处理

如果在 HTML 模板中设置了动态变量，那么在运行时，就可以通过 Context 实例设置这些变量。而在处理时，就会将具体的变量设置到模板中并输出。假如有以下的 HTML 模板：

```
<input type="text" th:value="${data.name}"/>
```

该 HTML 中使用了 th:value 属性来展示 data 对象的 name 属性，那么在处理模板时，就要加入相应的参数，请见代码清单 5-4。

代码清单 5-4：codes\05\5.2\first-th\src\main\java\org\crazyit\boot\c5\VarTest.java

```
public class VarTest {

    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
        templateEngine.setTemplateResolver(resolver);
        // 设置参数到 Context 中
        Context ctx = new Context();
        ctx.setVariable("data", new DataObject("angus"));
        String result = templateEngine.process("var.html", ctx);
        System.out.println(result);
    }
}
```

注意粗体字代码，其将 DataObject 设置到 Context 中。DataObject 是一个普通的 Java 类，只有一个 name 属性。运行代码清单 5-4，输出如下：

```
<input type="text" value="angus"/>
```

根据结果可知，Thymeleaf 将模板中的变量替换为了具体的值。

5.2.5 遍历集合

数据列表是最常用的功能之一，与前面的变量一样，只需要将数据集合放到 Context 中，就可以在模板里面使用 Thymeleaf 的标签进行遍历。下面，我们实现一个最简单的数据列表功能，其中模板处理的逻辑请见代码清单 5-5。

代码清单 5-5：codes\05\5.2\first-th\src\main\java\org\crazyit\boot\c5\IterationTest.java

```
public class IterationTest {

    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        // 创建解析器
        ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
        // 设置到模板引擎中
        templateEngine.setTemplateResolver(resolver);
        // 设置集合数据
        List<DataObject> datas = new ArrayList<DataObject>();
        datas.add(new DataObject("a"));
        datas.add(new DataObject("b"));
        Context ctx = new Context();
        ctx.setVariable("datas", datas);
        // 处理 classpath 下的 index.html 文件
        String result = templateEngine.process("iteration.html", ctx);
        System.out.println(result);
    }
}
```

在处理模板时，将集合放到 Context 中，变量名称为 datas，本例对应的模板文件为类路径下的 iteration.html，该文件的内容如下：

```
<table>
    <tr th:each="data : ${datas}">
        <td th:text="${data.name}">Angus</td>
    </tr>
</table>
```

在模板文件中，使用 th:each 属性进行数据遍历，节点变量为 data，在 td 中会使用 th:text 将 data 的 name 属性输出。运行代码清单 5-5，控制台输出如下：

```
<table>
    <tr>
        <td>a</td>
    </tr>
    <tr>
        <td>b</td>
    </tr>
</table>
```

5.2.6 设置前缀与后缀

可以在 TemplateResolver 中设置处理的前缀和后缀，那么在调用 process 方法时，只传入模板的名称即可，例如以下的运行代码：

```
        TemplateEngine templateEngine = new TemplateEngine();
        // 设置前缀与后缀
        ClassLoaderTemplateResolver resolover = new ClassLoaderTemplateResolver();
```

```
        resolover.setPrefix("/templates/");
        resolover.setSuffix(".html");
        templateEngine.setTemplateResolver(resolover);
        // 处理 /templates/index.html
        String result = templateEngine.process("index", new Context());
        System.out.println(result);
```

以上代码配置了前缀为“/templates/”，后缀为“.html”，而在处理时，会处理“前缀 + 模板名称 + 后缀”的模板文件，因此以上的代码会处理“/templates/index.html”模板文件。

除了以上的 API 外，Thymeleaf 还有许多其他的功能，我们会在后面的章节中陆续学到。这一节我们对 Thymeleaf 的 API 有个大概的了解即可。

5.3 在 Servlet 应用中整合 Thymeleaf

这一节，我们会在传统的 Web 应用中整合 Thymeleaf，并使用最原始的 Servlet 进行整合。实际上我们已经很少能接触到纯 Servlet 应用，这里使用它是希望通过整合过程，让大家更加了解 Thymeleaf 的 API，以便为后面的 Spring Boot 整合打下基础。

5.3.1 建立项目

新建一个普通的 Maven 项目，指定项目打包方式为 war，在 pom.xml 中加入以下依赖：

```
        <dependency>
            <groupId>org.thymeleaf</groupId>
            <artifactId>thymeleaf</artifactId>
            <version>3.0.9.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.5</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
```

由于我们要使用 JDK10，因此还要配置编译属性，所以为 pom.xml 添加以下配置：

```
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>10</maven.compiler.source>
        <maven.compiler.target>10</maven.compiler.target>
    </properties>
```

接下来，为项目加入 web.xml、log4j.properties 等文件，项目结构如图 5-1 所示。

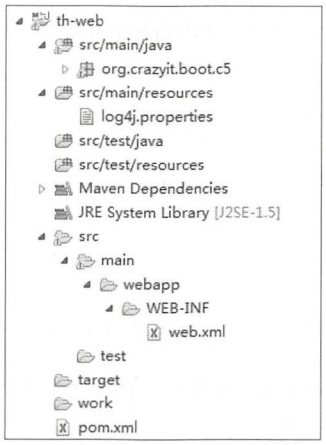


图 5-1 Web 项目的结构

在开发环境中，你可以使用 Eclipse 的 Tomcat 插件来运行本例的 Web 项目，也可以使用 Maven 的 Jetty 插件直接运行。关于 Eclipse 的 Tomcat 插件的使用方法，在此不展开讨论。如果要使用 Maven 的 Jetty 插件，则需要在 pom.xml 中加入以下配置：

```
<build>
    …
    <plugins>
        <plugin>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-maven-plugin</artifactId>
            <version>9.4.7.v20170914</version>
            <configuration>
                <webApp>
                    <contextPath>/${build.finalName}</contextPath>
                </webApp>
                <stopKey>CTRL+C</stopKey>
                <stopPort>8999</stopPort>
                <scanIntervalSeconds>10</scanIntervalSeconds>
                <scanTargets>
                    <scanTarget>src/main/webapp/WEB-INF/web.xml</scanTarget>
                </scanTargets>
            </configuration>
        </plugin>
    </plugins>
</build>
```

要启动 Web 项目，可以从命令行进入 pom.xml 所在的目录（本例为 codes\05\5.3\th-web），输入命令“mvn jetty:run”，即可启动 Jetty 容器，默认端口为 8080。

本例的代码中配置了 Jetty 插件，读者可根据具体情况，使用 Eclipse 的 Tomcat 插件或者 Maven 的 Jetty 插件来运行本例的应用。也可以使用命令行工具，进入 codes\05\5.3\th-web 目录，输入 mvn clean package，将项目打成一个 war 包，放到 Web 容器中运行。

5.3.2 配置模板引擎

对于一个应用，只需要配置一次 Thymeleaf 就可以了，每次请求都可以使用同一个 TemplateEngine 实例。我们可以使用单例模式。新建一个类，请见代码清单 5-6。

代码清单 5-6：codes\05\5.3\th-web\src\main\java\org\crazyit\boot\c5\WebApplication.java

```
public class WebApplication {

    private TemplateEngine templateEngine;

    private static WebApplication webApp;

    /*
     * 私有构造器
     */
    private WebApplication(ServletContext servletContext) {
        // 创建模板解析器
        ServletContextTemplateResolver templateResolver = new
ServletContextTemplateResolver(
                servletContext);
        // 设置模板模式为 HTML
        templateResolver.setTemplateMode(TemplateMode.HTML);
        // 设置前缀与后缀
        templateResolver.setPrefix("/WEB-INF/templates/");
        templateResolver.setSuffix(".html");
        templateResolver.setCharacterEncoding("UTF-8");
        this.templateEngine = new TemplateEngine();
        this.templateEngine.setTemplateResolver(templateResolver);
    }

    public static WebApplication getInstance(ServletContext servletContext) {
        if(webApp == null) {
            webApp = new WebApplication(servletContext);
        }
        return webApp;
    }

    public TemplateEngine getEngine() {
        return this.templateEngine;
    }
}
```

在 WebApplication 类的构造器中，对 TemplateEngine 进行了配置，并使用 ServletContextTemplateResolver 作为模板解析器，分别配置了模板模式、处理时的前缀与后缀。在 Servlet 中，只需要调用 WebApplication 的 getEngine 方法就可以获得 TemplateEngine 的实例。

注意：WebApplication 使用单例模式，在单例模式下，需要提供一个私有的构造器、一个返回自己实例的方法。

5.3.3 在 Servlet 中使用 Thymeleaf

编写一个简单的 Servlet，调用前一节 WebApplication 的 getEngine 方法得到 TemplateEngine，再进行数据设置和模板处理等，请见代码清单 5-7。

代码清单 5-7：codes\05\5.3\th-web\src\main\java\org\crazyit\boot\c5\UserServlet.java

```
public class UserServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
```

```
        WebApplication webApp = WebApplication.getInstance(this.getServletContext());
        // 设置用户数据
        Context ctx = new Context();
        ctx.setVariable("users", getUsers());
        // 获取模板引擎实例
        TemplateEngine engine = webApp.getEngine();
        String result = engine.process("user", ctx);
        response.setContentType("text/html;charset=UTF-8");
        // 输出渲染后的 HTML
        response.getWriter().print(result);
    }

    /*
     * 获取用户数据
     */
    private List<User> getUsers() {
        List<User> users = new ArrayList<User>();
        users.add(new User(1, "User A"));
        users.add(new User(2, "User B"));
        users.add(new User(3, "User C"));
        return users;
    }
}
```

其中粗体字代码用于获得 TemplatEngine 实例，之后，调用 process 方法处理模板，最后使用 HttpServletResponse 将结果输出。需要注意的是，由于我们在 WebApplication 中已经配置了模板引擎的前缀与后缀，因此在调用 process 方法时，只需要传入模板文件的名称（不需要.html），Thymeleaf 就会去“WEB-INF/templates”目录下寻找“user.html”文件。编写完 Servlet 类后，还要在 web.xml 中进行配置：

```
    <servlet-mapping>
        <servlet-name>UserServlet</servlet-name>
        <url-pattern>/users</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>UserServlet</servlet-name>
        <servlet-class>org.crazyit.boot.c5.UserServlet</servlet-class>
    </servlet>
```

5.3.4 编写模板文件

在 5.2 节中，我们已经初步接触了 Thymeleaf 的模板语法，所以我们知道，对于前面使用 Servlet 设置的数据集合，可以使用 th:each 进行遍历。模板文件的部分内容，请见代码清单 5-8。

代码清单 5-8：codes\05\5.3\th-web\src\main\webapp\WEB-INF\templates\user.html

```
<body>
    <table border="1" width="200px;" style="text-align: center;">
        <tr>
            <td width="30%">数据 ID</td>
            <td>用户名称</td>
        </tr>
        <tr th:each="user : ${users}">
            <td th:text="${user.id}">0</td>
            <td th:text="${user.name}">Angus</td>
```

```
        </tr>
    </table>
</body>
```

将我们的项目放到 Tomcat 等服务器中运行，启动服务器后，在浏览器中访问 http://localhost:8080/th/users，浏览器的输出如图 5-2 所示。



图 5-2　运行结果

可以看到，我们已经可以在 Servlet 应用中使用 Thymeleaf 了。实际上，我们可以对前面的程序进行进一步的重构、优化，有需要的读者可以根据实际情况来改进，只要熟练掌握模板引擎的设置过程、模板的处理过程即可。

5.4　在 Struts2 中整合 Thymeleaf

Struts2 曾经是最流行的MVC框架，相信目前仍有不少的项目在使用它，但随着Spring Boot 的兴起，越来越多的开发者投入到了 Spring MVC 的怀抱。虽然 Struts2 有点日薄西山的意思，但为了表现 Thymeleaf 的灵活性，本节的示例会整合 Struts2 与 Thymeleaf。笔者希望，不只在 Spring MVC 的项目中能见到 Thymeleaf 的身影。

5.4.1　建立与配置 Struts2 项目

本例所使用的 Struts2 版本为 2.5.16。新建一个 Maven 项目，加入依赖并配置属性：

```
    <dependencies>
        <dependency>
            <groupId>org.apache.struts</groupId>
            <artifactId>struts2-core</artifactId>
            <version>2.5.16</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.10.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.10.0</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.5</version>
        </dependency>
        <dependency>
```

```
            <groupId>com.github.a-pz</groupId>
            <artifactId>struts2-thymeleaf3-plugin</artifactId>
            <version>1.0.5-RELEASE</version>
        </dependency>
    </dependencies>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>10</maven.compiler.source>
        <maven.compiler.target>10</maven.compiler.target>
    </properties>
```

与 5.3 节的 Servlet 项目一致，在该项目的 pom.xml 中，同样加入了 Maven 的 Jetty 插件。我们可以从命令行进入 pom.xml 所在目录（本例为 codes\05\5.4\th-struts），运行“mvn jetty:run”命令，来直接启动 Jetty 服务器，也可以直接打成 war 包，放到 Tomcat 等服务器中运行。在项目的 web.xml 中配置 Struts2 的过滤器，请见代码清单 5-9。

代码清单 5-9：codes\05\5.4\th-struts\src\main\webapp\WEB-INF\web.xml

```
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.filter.
StrutsPrepareAndExecuteFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

除了配置 web.xml 外，还需要在 src/main/resources 下加入 log4j2.xml 与 struts.xml。为了测试配置，在 src/main/webapp 目录下加入了一个 test.jsp 文件，启动服务器后，访问 http://localhost:8080/ths/test.jsp，可以看到该页面的内容。

```
public class TestAction extends ActionSupport {

    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

5.4.2 配置 Struts Action

配置一个 Action 类，用于测试 Struts2 环境。新建 Action 类，请见代码清单 5-10。

代码清单 5-10：codes\05\5.4\th-struts\src\main\java\org\crazyit\boot\c5\TestAction.java

```
public class TestAction extends ActionSupport {

    public String execute() throws Exception {
        System.out.println("struts2 action");
        return SUCCESS;
    }
}
```

在 struts.xml 文件中配置 Action，如以下配置：

```
<struts>
```

```
    <constant name="struts.devMode" value="true" />
    <package name="test" extends="struts-default">
        <action name="test" class="org.crazyit.boot.c5.TestAction">
            <result name="success">/test.jsp</result>
        </action>
    </package>
</struts>
```

完成配置后启动服务器，在浏览器中访问 http://localhost:8080/ths/test.action，则可以看到页面与控制台的输出。这里使用的是最传统的 Struts Action 配置方式，实际上 Struts2 可以支持使用注解的方式来配置 Action，但本书的重点不是 Struts2，因此不深入研究 Struts2 的使用。

5.4.3 整合 Themeleaf

在 Struts2 中整合 Thymeleaf，需要编写 Struts2 的插件，笔者在 Github 上找到一个 Struts2-Thymeleaf3 插件项目，可以使用它整合这两个框架。在 pom.xml 文件中加入以下依赖：

```
        <dependency>
            <groupId>com.github.a-pz</groupId>
            <artifactId>struts2-thymeleaf3-plugin</artifactId>
            <version>1.0.5-RELEASE</version>
        </dependency>
```

需要说明的是，该依赖已经被发布到 Maven 的官方仓库，因此并不需要额外配置仓库。新建一个 Action 类，用于设置数据，请见代码清单 5-11。

代码清单 5-11：codes\05\5.4\th-struts\src\main\java\org\crazyit\boot\c5\UserAction.java

```
public class UserAction extends ActionSupport {

    private List<User> users;

    @Override
    public String execute() throws Exception {
        this.users = createUsers();
        return SUCCESS;
    }

    /*
     * 获取用户数据
     */
    private List<User> createUsers() {
        List<User> users = new ArrayList<User>();
        users.add(new User(1, "User A"));
        users.add(new User(2, "User B"));
        users.add(new User(3, "User C"));
        return users;
    }

    public List<User> getUsers() {
        return users;
    }

    public void setUsers(List<User> users) {
        this.users = users;
    }
}
```

在 UserAction 中，只是简单地设置了用户数据集合，该集合将会在 HTML 模板中展示。修改 struts.xml，加入 Thymeleaf 及 Action 的配置：

```
<constant name="struts.thymeleaf.templateMode" value="HTML" />
<constant name="struts.thymeleaf.encoding" value="UTF-8" />
<constant name="struts.thymeleaf.prefix" value="/WEB-INF/templates/" />
<constant name="struts.thymeleaf.suffix" value=".html" />
<package name="th" extends="struts-thymeleaf">
    <action name="user" class="org.crazyit.boot.c5.UserAction">
        <result name="success" type="thymeleaf">user</result>
    </action>
</package>
```

在 struts.xml 中加入了 4 个 constant 节点，根据配置项可知，最终会调用 Thymeleaft 的 API 来配置解析器。在配置 Action 时，返回的 result 类型为“thymeleaf”，该类型被定义在我们引用的插件中。根据我们配置的前缀与后缀，当返回“success”时，会到“src/main/webapp/WEB-INF/templates”目录下寻找 user.html 模板。该模板内容较为简单，只是使用 Thymeleaf 的语法对 User 集合进行遍历与输出：

```
<tr th:each="user : ${users}">
    <td th:text="${user.id}">0</td>
    <td th:text="${user.name}">Angus</td>
</tr>
```

启动服务器，访问 http://localhost:8080/ths/user，可以看到浏览器的输出。在 Struts2 中整合 Thymeleaf 较为简单，基本上不需要接触 Thymeleaf 的 API，这是由于我们所使用的 Struts2 Thymeleaf 插件，已经帮我们完成了 Thymeleaf 的配置、模板处理等工作。

5.5 在 Spring Boot 中整合 Thymeleaf

在前面的章节中，我们构建 Web 应用需要做一些固定的工作，而使用 Spring Boot 就较为简单，为项目加入 Spring Boot 提供的 Thymeleaf Starter 模块，再配置一下 Thymeleaf 就可以了。这一节我们讲解 Spring Boot 与 Thymeleaf 的整合。

5.5.1 建立与配置项目

新建一个 Maven 项目，pom.xml 文件的主要内容请见代码清单 5-12。

代码清单 5-12：codes\05\5.5\th-boot\pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.1.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
```

```
        </dependency>
    </dependencies>
```

接下来，为 Spring Boot 项目加入配置文件，在 src/main/webapp 下新建 application.properties 文件，内容如下：

```
spring.thymeleaf.mode=HTML
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
```

以上的几个配置，我们在前面章节已经看到过多次，一开始使用 API 来配置，在整合 Struts2 时将它们配置在 struts.xml 文件中，在 Spring Boot 中又将这些配置放在 application.properties 中。相信读者已经很熟悉以上这些配置，不过需要特别说明的是“spring.thymeleaf.prefix”配置，该模板文件会被放在类路径下的 templates 目录中，默认放在 src/main/resources 目录下。在配置文件 application.properties 中，可以修改的 Thymeleaf 配置主要有以下几项。

- spring.thymeleaf.cache：是否打开模板配置，默认为 true。
- spring.thymeleaf.check-template：是否在处理模板前检查其是否存在，默认为 true。
- spring.thymeleaf.check-template-location：是否检查模板位置，默认为 true。
- spring.thymeleaf.content-type：模板的 content-type 值，默认为 text/html。
- spring.thymeleaf.enabled：是否开启视图解析，默认为 true。
- spring.thymeleaf.encoding：模板的编码，默认为 UTF-8。
- spring.thymeleaf.excluded-view-names：要排除的模板名称。
- spring.thymeleaf.mode：模板模式，默认为 HTML5。
- spring.thymeleaf.prefix：模板前缀，默认为 classpath:/templates/，用于创建模板的 URL。
- spring.thymeleaf.suffix：模板后缀，默认为.html，用于创建模板 URL。“前缀 + 模板名称 + 后缀”即可定位到具体的模板。
- spring.thymeleaf.template-resolver-order：默认模板解析器的执行顺序。
- spring.thymeleaf.view-names：要解析的模板名称，用逗号隔开。

以上执行简单的两步就完成了 Spring Boot 与 Thymeleaf 的整合，实际上都不需要配置 application.properties，使用默认值即可，换言之，只需要为项目加入“spring-boot-starter-thymeleaf”依赖，就能完成整合。这相对于 Struts2 与 Thymeleaf 的整合，显得简单多了。

5.5.2 编写控制器与模板

我们实现一个简单的功能，来测试 Spring Boot 与 Thymeleaf 的整合情况。编写启动类与控制器，并设置数据，请见代码清单 5-13。

代码清单 5-13：codes\05\5.5\th-boot\src\main\java\org\crazyit\boot\c5\ThyApp.java

```
@SpringBootApplication
@Controller
public class ThyApp {

    public static void main(String[] args) {
        SpringApplication.run(ThyApp.class, args);
    }

    @RequestMapping("/users")
```

```
    public String listUsers(Model model) {
        model.addAttribute("users", queryUsers());
        return "user";
    }

    private List<User> queryUsers() {
        List<User> users = new ArrayList<User>();
        users.add(new User(1, "userA"));
        users.add(new User(2, "userB"));
        return users;
    }
}
```

控制器中的 listUsers 方法直接返回“user”字符串，根据 Thymeleaf 的配置，其会到 templates 目录下寻找 user.html 模板，本例的 user.html 模板只是简单地对 User 集合进行遍历：

```
        <tr th:each="user : ${users}">
            <td th:text="${user.id}">0</td>
            <td th:text="${user.name}">Angus</td>
        </tr>
```

直接运行 ThyApp 的 main 方法，在浏览器中访问：http://localhost:8080/users，就可以看到输出结果。

模板与控制器的代码与前面的例子无异，我们关键要掌握 Thymeleaf 与各个框架的整合方法。最重要的是，我们要知道，最终会调用 Thymeleaf 的 API 来对模板引擎进行配置。

5.5.3 静态资源

在默认情况下，Spring Boot 会到以下的目录读取静态资源。

- classpath:/META-INF/resources/
- classpath:/resources/
- classpath:/static/
- classpath:/public/

例如有一个 common.css 文件，我们一般会将其放到 src/main/resources/static 目录下。在 Thymeleaf 模板中，可以使用链接表达式来使用这个 CSS 文件：

```
<link rel="stylesheet" type="text/css" th:href="@{/common.css}" />
```

以上的 4 个目录都可以用于存放 CSS、JS、图片等静态资源，如果需要修改静态资源的存放位置，则可以修改 application.properties（或 yml）的 spring.resources.static-locations 属性，多个目录之间可用逗号隔开。

5.6 本章小结

本章初步介绍了 Thymeleaf 这个模板引擎，介绍了 Thymeleaf 最基本的一些 API，例如模板引擎的创建、配置及模板处理等。掌握了这些基本 API 的使用，再学习本章后面的内容就很轻松了。从 5.3 节开始，讲解了 Thymeleaf 在 Web 应用中的整合，包括与 Servlet 应用的整合、与 Struts2 的整合、与 Spring Boot 的整合。虽然目前使用 Servlet 或者 Struts2 的项目较少，但笔者觉得，Thymeleaf 作为一个成熟的模板引擎，可以被应用在任何的 Web 项目中，而不仅仅是 Spring Boot 中。当然，Spring Boot 作为本书的重点，如何将其与 Thymeleaf 整合，也是我们必须掌握的知识。

CHAPTER

6

第6章 Thymeleaf 对象的使用

本章要点

- Thymeleaf 文本处理
- 基本对象
- 数字与字符串对象
- 日期与集合对象
- 其他类型对象

在前面章节我们学习了 Spring Boot 的使用，并了解了模板引擎 Thymeleaf，而且知道如何将 Thymeleaf 整合到我们的项目中。但仅懂得这些明显是不够的，为了在项目中更好地应用 Thymeleaft，从本章开始，我们将学习 Thymeleaf 的具体语法。

本章的重点就是学习语法，因此涉及语法的大部分示例，都直接使用代码来运行 Thymeleaf 模板，而不会放到 Web 环境中运行，这一点需要读者注意。

6.1 文本处理

在学习 Thymeleaf 的对象前，我们先学习如何设置、读取变量，以及如何在模板中显示这些变量。建立一个普通的 Maven 项目，引入 Thymeleaf 的依赖：

```
        <dependency>
            <groupId>org.thymeleaf</groupId>
            <artifactId>thymeleaf</artifactId>
            <version>3.0.9.RELEASE</version>
        </dependency>
```

6.1.1 外部文本输出

我们来了解一下 message 这个概念，在 Thymeleaf 中，外部的文本片断通常被称为 message，为了能与 Java 的消息做区分，本书将其称为“文本信息”，它们通常会被保存在 properties 文件里面，因此每一个文本信息都有一个 key 作为标识。在 Thymeleaf 的模板中，只需要使用“#{template.key}”这样的表达式（注意是#号），就可以将配置好的文本输出，本书称这种表达式为文本表达式。新建模块，输出普通的 HTML 语句，请见代码清单 6-1。

代码清单 6-1：codes\06\6.1\gm-text\src\main\resources\externalized.html

```
<span th:text="#{page.myText}">This is prototype text.</span>
```

在该模板中使用了 th:text 属性来读取“page.myText”属性，在默认情况下，会到类路径下寻找 externalized.properties 文件，并将 properties 文件中的“page.myText”替换至模板中，本例的 properties 文件内容如下：

```
page.myText=This is config text.
```

编写处理类，调用 Thymeleaf 的 API 来处理模板，请见代码清单 6-2。

代码清单 6-2：codes\06\6.1\gm-text\src\main\java\org\crazyit\boot\c6\ExternalizedTest.java

```
public class ExternalizedTest {

    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        // 创建解析器
        ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
        // 设置到模板引擎中
        templateEngine.setTemplateResolver(resolver);
        // 处理类路径下的 externalized.html 文件
        String result = templateEngine.process("externalized.html", new Context());
        System.out.println(result);
    }
}
```

运行代码 6-2，输出结果如下：

```
<span>This is config text.</span>
```

根据结果可知，在 externalized.properties 文件中配置的文本，已经被替换到模板中。文本表达式“#{key}”的效果等同于“${#messages.msg(key)}”，#messages 是 Thymeleaf 的内置对象之一，后面章节会讲解它。

注意：在 Spring Boot 环境下有另外的 properties 文件读取规则，在相关章节讲述。

6.1.2 文本国际化

前面提到，Spring Boot 默认会到类路径下读取“模板名称.properties”属性文件（Spring Boot 环境下有另外的规则），并在文件中寻找相应的 key，这样的说法并不严谨，实际上，Thymeleaf 会按照一定的规则与顺序去寻找 properties 文件。Thymeleaf 会调用 java.util.Locale 的 getLanguage、getCountry 与 getVariant 方法，分别得到语言（language）、国家（country）与方言（variant）属性，再根据这三个属性，按照顺序读取 properties 文件。假设模板名称为 home，语言为 zh，国家为 CN，方言为 ma，则读取文件的顺序如下：

1. home_zh_CN-ma.properties，注意 ma 前是“-”，其他均是下画线
2. home_zh_CN.properties
3. home_zh.properties
4. home.properties

读取哪一个文件，关键是看设置了什么样的 java.util.Locale 实例，请见以下的模板：

```
<span th:text="#{i18n.myKey}">Prototpye text</span>
```

该模板的文件名称为 i18n.html，我们提供两个 properties 文件，分别用来设置不同的方言，请见代码清单 6-3。

代码清单 6-3：codes\06\6.1\gm-text\src\main\resources\i18n_zh_CN-ma.properties

```
i18n.myKey=郁闷了兄弟
```

codes\06\6.1\gm-text\src\main\resources\i18n_zh_CN-db.properties

```
i18n.myKey=扎心了老铁
```

接下来，编写代码来处理模板，处理两次，分别为它们设置不同的 Locale，请见代码清单 6-4。

代码清单 6-4：codes\06\6.1\gm-text\src\main\java\org\crazyit\boot\c6\I18nTest.java

```
TemplateEngine templateEngine = new TemplateEngine();
// 创建解析器
ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
// 设置到模板引擎中
templateEngine.setTemplateResolver(resolver);
// 处理类路径下的 html 文件
Context ctx = new Context();
// 本机默认的 Local 为 zh、CN
Locale maLocale = new Locale("zh", "CN", "ma");
ctx.setLocale(maLocale);
String result = templateEngine.process("i18n.html", ctx);
```

```
        System.out.println("使用 zh_CN-ma.properties 文件: " + result);
        // ============ 使用新的 Locale ============
        Context ctx2 = new Context();
        Locale dbLocale = new Locale("zh", "CN", "db");
        ctx2.setLocale(dbLocale);
        result = templateEngine.process("i18n.html", ctx2);
        System.out.println("使用 zh_CN-db.properties 文件: " + result);
```

在代码清单 6-4 中，创建了两个 Context，这两个 Context 被设置了不同的 Locale 实例，最终 Thymeleaf 会根据不同的 Locale 来读取不同的 properties 文件。运行代码清单 6-4，输出如下：

```
使用 zh_CN-ma.properties 文件: <span>郁闷了兄弟</span>
使用 zh_CN-db.properties 文件: <span>扎心了老铁</span>
```

通过以上的处理，我们很容易就实现了文本信息的国际化。在这里我们主要掌握 Thymeleaf 对 properties 文件的读取规则。

6.1.3 处理转义文本

在 properties 文件中，如果值里面存在 HTML 节点，则在使用 th:text 属性输出时，会将 HTML 节点处理为转义字符并输出，例如以下的文本值：

```
data.key=你好, <span style="color: red">疯狂 Java</span>
```

使用 th:text 输出，结果如下：

```
你好, &lt;span style=&quot;color: red&quot;&gt;疯狂 Java&lt;/span&gt;
```

这显然不是我们想要的结果，而这个字符串，浏览器也无法处理。我们可以在模板中使用 th:utext 来解决该问题，针对同一段文本，使用 th:utext 后，结果如下：

```
你好, <span style="color: red">疯狂 Java</span>
```

以上的这段 HTML 就是我们所期望的结果。因此，如果在 properties 文件中存在特别的字符，可直接使用 th:utext 属性。

6.1.4 读取变量

在上一章的整合例子中，我们已经尝试过在模板中获取变量，只需要将变量设置到 Context 中即可。如我们在代码清单 6-5 中设置了变量。

代码清单 6-5：codes\06\6.1\gm-text\src\main\java\org\crazyit\boot\c6\VarTest.java

```
        TemplateEngine templateEngine = new TemplateEngine();
        // 设置模板解析器
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        Context ctx = new Context();
        ctx.setVariable("userName", "Angus");
        String result = templateEngine.process("var.html", ctx);
        System.out.println(result);
```

以上例子对应的模板内容如下：

```
<span th:text="${userName}">This is prototype text.</span>
```

在模块中显示文本消息（message）使用“#{...}”表达式（文本表达式），而此处要显示的是变量值，因此使用“${...}”表达式，本书称之为变量表达式。

在变量表达式的中间放的是 OGNL 表达式，使用过 WebWork 或者 Struts2 的朋友可能对此比较熟悉，OGNL 是一种表达式语言，使用它可以设置与获取对象属性。本例的“userName”只是简单的字符串变量，我们甚至可以获取对象的属性，例如“${user.name}”。除了“#{...}”与“${...}”这两个表达式外，Thymeleaf 还有其他的表达式，我们会在后面章节中讲述。

6.1.5 在文本信息中使用变量

如前面章节所述，可以直接在模板中使用“${...}”表达式来访问变量，除此之外，还可以将“${...}”放到“#{...}”表达式中，用于替换文本信息。在 Context 中设置一个 User 对象（id 与 name 属性），详细请见代码清单 6-6。

代码清单 6-6：codes\06\6.1\gm-text\src\main\java\org\crazyit\boot\c6\VisitVarTest.java

```
TemplateEngine templateEngine = new TemplateEngine();
// 设置模板解析器
templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
Context ctx = new Context();
User user = new User();
user.setId(1);
user.setName("Angus");
ctx.setVariable("user", user);
String result = templateEngine.process("visit_var.html", ctx);
System.out.println(result);
```

在该模板中，如果要将用户名、id 传递到文本信息（properties）中，则需要在“#{...}”中嵌套使用“${...}”语句，模板内容如下：

```
<div th:text="#{data.key(${user.name}, ${user.id})}">Prototype content</div>
```

对应的 properties 文件内容如下：

```
data.key=你好，{0}，你的 id 是：{1}
```

模板文件在使用 th:text 属性、指定 data.key 时，传入了 user.name 与 user.id，properties 文件根据传入的参数索引（从 0 开始）来使用对应的参数，本例中索引为 0 的参数是 user.name，索引为 1 的参数是 user.id。运行代码清单 6-6，处理模板后，输出结果如下：

```
<div>你好，Angus，你的 id 是：1</div>
```

6.1.6 其他读取变量的方式

在以上的例子中，会调用 User 的 getter 方法获取对应的类属性。除了获取单个对象的属性外，OGNL 还支持其他获取对象（或属性）的方法，如代码清单 6-7，为 Context 设置了不同的数据。

代码清单 6-7：codes\06\6.1\gm-text\src\main\java\org\crazyit\boot\c6\MixVarTest.java

```
Context ctx = new Context();
// 设置 User 的 Parent 属性
User user = new User();
Parent parent = new Parent();
parent.setAge(10);
```

```
        user.setParent(parent);
        ctx.setVariable("user", user);
        // 设置集合变量
        List<User> users = new ArrayList<User>();
        users.add(new User(1, "Angus.L"));
        users.add(new User(2, "Paris.L"));
        ctx.setVariable("users", users);
        // 设置 Map 变量
        Map<String, User> userMap = new HashMap<String, User>();
        userMap.put("key1", new User(1, "Angus.M"));
        userMap.put("key2", new User(2, "Paris.M"));
        ctx.setVariable("userMap", userMap);
        // 设置数组
        User[] userArr = new User[]{new User(1, "Angus.A"), new User(2, "Paris.A")};
        ctx.setVariable("userArr", userArr);
```

在模板中，可使用不同的方式获取相应的值。

- **获取变量中的属性：**
- **获取 List 中的第 1 个元素：**
- **获取 Map 中的元素：**
- **获取数组中的第 2 个元素：**

如果获取的对象是 null，则获取它的类属性（例如 List 中的第 1 个 User 是 null，执行 user[0].name）时，会抛出类似的异常：“org.thymeleaf.exceptions.TemplateProcessingException: Exception evaluating OGNL expression”，注意并不是 NullPointerException。

注意：如果 Web 应用使用了 Spring MVC，则 OGNL 将会被替换为 SpringEL，因此在整合 Spring Boot 之后，默认会使用 SpringEL。

6.1.7 链接表达式

在 Web 环境下，URL 的处理较为重要，Thymeleaf 对此提供了链接表达式（@{...}），我们可以使用其设置 URL。该表达式支持绝对路径与相对路径。假设 context-path 为“oa”，则我们可以在链接表达式中使用下表中所列的几种格式的 URL。

	表达式	处理结果	访问路径
绝对路径	@{http://www.qq.com}	http://www.qq.com/	www.qq.com
页面相对路径（无字符开头）	@{order/details}	order/details	http://ip:port/oa/order/details
上下文相对路径（/开头）	@{/order/details}	/oa/order/details	http://ip:port/oa/order/details
服务器相对路径（~/开头）	@{~/order/details}	/order/details	http://ip:port/order/details
协议相对路径（//开头）	@{//order/details}	//order/details	order/details

以上展示了几种 URL 的设置方式，特别是相对路径的设置，大家需要留意每一种的特别之处，例如使用上下文相对路径，要使用“/”开头。另外也对比了几种路径设置的处理结果及访问路径，我们可根据实际应用的需求，来决定如何配置 URL。

链接表达式支持设置参数，例如可以使用以下的代码来设置 URL 参数：

```
    <a href="view.html" th:href="@{/user/details(id=${user.id})}" th:text=
"${user.name}"></a>
    处理结果：<a target="_blank" href="/order/details?id=1">userA</a>
```

如果存在多个路径参数，则可以在括号中使用逗号隔开，处理后 Spring Boot 会自动帮我们添加“&”：

```
    <a  href="view.html"  th:href="@{/user/details(id=${user.id},  name=${user.name})}"
th:text="${user.name}"></a>
    处理结果：<a href="/my-oa/user/details?id=2&name=userB">userB</a>
```

可以这样来使用路径参数：

```
    <a    href="view.html"    th:href="@{/user/{userId}/details(userId=${user.id})}"
th:text="${user.name}"></a>
```

除了以上的 a 节点之外，其他有 URL 的地方，都可以使用这个表达式，例如以下 HTML 片断：

```
    <link  href="../css/test.css"  th:href="@{/css/test.css}"  rel="styleshee"  type=
"text/css" />
    <script  type="text/javascript"  src="../js/test.js"  th:src="@{/js/test.js}">
</script>
```

提示：上下文相对路径（/开头的 URL）只能在 Web 环境下使用，在其他环境下使用会得到以下异常：org.thymeleaf.exceptions.TemplateProcessingException: Link base "/user/details" cannot be context relative。

6.2　基本对象

Thymeleaf 中有许多内置对象，这些内置对象可以帮我们在模板中实现各种功能，例如获取 request、session 里面的值，还可以处理字符串、日期等数据。这一节，我们学习几个内置的基本对象。

注意：如果你想查看源代码来了解 Thymeleaf 的内置对象，则可以查看 StandardExpressionObjectFactory 这个类。

6.2.1　基本对象概述

在 JSP 页面中，我们可以很轻松地获取几个内置对象，例如 request、session 等。在 Thymeleaf 的模板中，也提供了类似的内置对象，如下面几个是其基本的内置对象。

- #ctx：表示模板引擎的全局上下文对象。
- #vars：同样返回模板引擎上下文对象，一般直接使用#ctx 即可。
- #locale：在全局上下文中维护的 java.util.Locale 对象。
- #request：表示 HttpServletRequest 对象。
- #response：表示 HttpServletResonse 对象。
- #session：表示 HttpSession 对象。
- #servletContext：表示 Servletcontext 对象。

在模板中，可以使用“${#ctx.xxx}”这样的表达式来调用对象的方法，或者获取对象的属性值。需要注意的是，request、response、session 与 servletContxt 都是只能在 Web 环境下使用。下面详细讲述这几个基本对象的使用。

6.2.2 #ctx 对象

#ctx 表示模板引擎的全局上下文，如果在一般的环境下，其会返回一个 EngineContext 的实例，如果在 Web 环境下，则会返回一个 WebEngineContext 的实例，虽然它们都是 IContext 接口的实现类，但笔者建议，使用实现类将它们区分开。在此，大家只需要知道，#ctx 表示模板引擎上下文，它保存模板引擎的变量。

注意，#ctx 与另外一个 IContext 对象的区别。Thymeleaft 不管是否在 Web 环境下，最终都会调用 TemplateEngine 的 process 方法处理模板。该方法要求传入一个 IContext 的实例，这个 IContext 实例可以被看作一个请求上下文，与#ctx 是两个不同的实例，它们的实现类也不一样，读者别把这两者搞混了。

在模板中获取的#ctx 对象，可以直接以下面的方式来获取变量：

```
${#ctx.变量名称}
```

除了#ctx 对象外，还可以使用#locale 对象，它表示上下文对象中所维护的 java.util.Locale 对象，例如，我们可以在模板中使用以下方式来获取语言：

```
${#locale.language}
```

注意：在一般环境下，使用#vars、#ctx 与#root 都返回同样的对象（EngineContext），推荐使用#ctx。而在 Spring MVC 环境下，模板中的#vars 与#ctx 是同一个对象（WebEngineContext），#root 则会被 Spring 处理，返回 SPELContextMapWrapper 实例，要得到上下文，同样使用#ctx 即可。

6.2.3 Web 对象

我们平时使用最多的三个 Web 对象是：request、session 与 servletContext，这些对象只能在 Web 环境中使用。我们建立一个 Spring Boot 项目来测试这几个对象，Spring Boot 与 Thymeleaf 的整合，请见上一章的内容。在控制器中分别设置不同的参数，请见代码清单 6-8。

代码清单 6-8：codes\06\6.2\basic-object\src\main\java\org\crazyit\boot\c6\BasicApp.java

```
@RequestMapping("/test")
public String test(HttpServletRequest request, HttpSession session) {
    request.setAttribute("name1", "Angus");
    session.setAttribute("name2", "Paris");
    request.getServletContext().setAttribute("name3", "Test");
    return "test";
}
```

对应的模板内容请见代码清单 6-9。

代码清单 6-9：codes\06\6.2\basic-object\src\main\resources\templates\test.html

```
<div th:text="${#request.getAttribute('name1')}"></div>
<div th:text="${#session.getAttribute('name2')}"></div>
<div th:text="${#servletContext.getAttribute('name3')}"></div>
```

在模板中分别使用三个 Web 对象的 getAttribute 方法来输出值，除了该方法外，还可以使用其他方法，例如可以调用#request 的 getMethod 方法，获取 HTTP 的请求方法。在使用这些对象时，直接把它们当作相应的 Servlet 对象即可。运行本例的启动类，使用浏览器访问：http://localhost:8080/test，则可以看到结果。

6.2.4 内置变量

在 Web 环境中，Thymeleaf 提供了几个内置变量供我们使用：param、session 与 application，它们分别可以访问请求参数、session 属性与 application 属性。例如可以使用这样的语句来获取 session 中的属性：${session.userName}。细心的读者可能会发现，内置变量中怎么没有 request？实际上，request 的所有属性，都可以直接使用${myAttribute}来访问。代码清单 6-10 在控制器中设置各个属性。

代码清单 6-10：codes\06\6.2\basic-object\src\main\java\org\crazyit\boot\c6\BasicApp.java

```
@RequestMapping("/np")
public String np(@RequestParam String userName,
      @RequestParam String passwd, HttpServletRequest request) {
   request.setAttribute("reqData", "reqData");
   request.getSession().setAttribute("sesData", "sesData");
   request.getServletContext().setAttribute("appData", "appData");
   return "np";
}
```

对应的模板文件内容如下：

```
<div th:text="${param.userName}"></div>
<div th:text="${param.passwd}"></div>
<div th:text="${reqData}"></div>
<div th:text="${session.sesData}"></div>
<div th:text="${application.appData}"></div>
```

运行启动类，在浏览器中访问：http://localhost:8080/form，填写表单并提交，之后即可看到结果。这里需要提醒的是，内置对象与内置变量是两个概念，使用“${#对象}”形式的表达式来使用内置对象，而使用内置变量则不需要使用“#”。

6.2.5 使用基本对象的注意事项

在使用 Thymeleaf 的基本对象时，需要注意以下几个问题：

- 在一般环境中（OGNL 表达式），使用“${#ctx.locale}”这样的表达式并不能获得 Locale 对象，如果这样来调用：“${#ctx.locale.country}”，则会得到异常信息：ognl.OgnlException: source is null for getProperty(null, "country")，这是由于拿到的 Locale 对象为 null 的缘故。
- 在 Spring MVC 环境下（SpringEL 表达式），使用“${#ctx.session}”这样的表达式可以获取上下文中的内置变量，目前可以拿到前面所述的三个内置变量：param、session 与 application。但是要注意，使用这样的方式是无法获取 rquest、locale 等“对象”的，因为 request、locale 并不是内置变量。如果使用“${#ctx.locale.country}”这样的语句，则会得到这样的异常信息：SpelEvaluationException: EL1007E: Property or field 'country' cannot be found on null，因为使用这样的方式，拿到的 Locale 对象同样为 null。如果想通过#ctx 对象拿到 Local，则可以使用这样的表达式：“${#ctx.#locale}”，这样使用 Loacle 对象，就不会抛出异常。
- 官方文档在介绍三个内置变量 param、session 与 application 时，关于相关的 API 让我们查阅 WebRequestParamsVariablesMap、WebSessionVariablesMap 与 WebServletContext-VariablesMap 这三个类，实际上当前版本的 API 中并没有这三个类，笔者查看源码时发现，这三个内置变量实际上是三个自定义的 Map。我们在使用时，直接把它们当作

Map 来使用即可，例如直接调用 size 方法：“${param.size()}”，或者直接根据 key 获取 value：“${param.myKey}”。

我们在使用基本对象时需要注意以上的问题，并尽可能避免一些特殊的写法，例如要使用 Locale 对象，就直接使用“${#locale}”语法，而不要通过这样的方式获取：“${#ctx.locale}”。

6.3 数字对象

Thymeleaf 主要使用 org.thymeleaf.expression.Numbers 这个类来处理数字，在模板中使用“#numbers”来表示这个对象。这一节，我们学习在模板中处理数字的方法，为了简单起见，我们直接在 main 方法中测试这些对象的使用，不需要运行 Web 环境。

6.3.1 整数格式化

#numbers 对象提供了四个格式化整数的方法：formatInteger、arrayFormatInteger、listFormatInteger 与 setFormatInteger，根据名字可知，后面三个方法分别用于处理数组、List 与 Set，而第一个方法则用于处理单个整数。这四个方法的描述如下。

- formatInteger(number, digits)：第一个参数是处理的整数，第二个参数用于设置最少的整数位数。
- arrayFormatInteger (numbers, digits)：与 formatInteger 类似，但要传入数组，也会返回数组，例如可以在参数中定义数组：“${numbers.arrayFormatInteger({10, 20}, 5)}”。
- listFormatInteger (numbers, digits)：与 formatInteger 类似，传入数字 List，返回处理后的 List，如果参数是数组类型，想转换为 List，可以使用“#lists.toList(array)”进行转换。
- setFormatInteger (numbers, digits)：与 formatInteger 类似，传入数字 Set，返回处理后的 Set，如果参数是数组类型，想转换为 Set，可以使用“#sets.toSet(array)”进行转换。

总结：给什么类型的数据，它就会返回什么类型的数据。

新建一个 main 方法来进行测试，将各类型的数字设置到 Context 中，请见代码清单 6-11。

代码清单 6-11：codes\06\6.3\numbers\src\main\java\org\crazyit\boot\c6\IntegerTest.java

```
TemplateEngine templateEngine = new TemplateEngine();
templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
Context ctx = new Context();
Double[] arr = new Double[]{10D, 100.9};
List list = Arrays.asList(arr);
Set set = new HashSet(list);
ctx.setVariable("arr", arr);
ctx.setVariable("list", list);
ctx.setVariable("set", set);
String result = templateEngine.process("int.html", ctx);
System.out.println(result);
```

数组、List 与 Set 里面的元素都是 10、100.9，对应的 int.html 模板内容如下：

```
formatInteger: <div th:text="${#numbers.formatInteger(10.99, 5)}"></div>
<div th:each="num : ${#numbers.arrayFormatInteger(arr, 5)}">
   <div th:text="${num}"></div>
```

```
    </div>
    <div th:each="num : ${#numbers.listFormatInteger(list, 5)}">
        <div th:text="${num}"></div>
    </div>
    <div th:each="num : ${#numbers.setFormatInteger(set, 5)}">
        <div th:text="${num}"></div>
    </div>
```

在调用这四个方法时，都设置了最小的整数位数为 5。另外还要注意，你处理的是集合数据，同样会返回集合，因此要在模板中使用 th:each 属性对返回的结果进行遍历。运行代码清单 6-11，输出结果如下：

```
formatInteger: <div>00011</div>
<div>
    <div>00010</div>
</div>
<div>
    <div>00101</div>
</div>
...省略 List 与 Set 的输出
```

根据结果可知，如果输入的整数位数不足 5，则会加上“0”补足，如果输入的数字存在小数点，则会进行取整，四舍五入到个位，大家在使用时注意这两个细节即可。

这四个方法，每一个都存在重载方法，调用时如果传入第三个参数，则该参数将会用来作为“千位分隔符”，如果不设置第三个参数，则表示不使用千位分隔符，如以下代码设置分隔符为逗号：

```
${#numbers.formatInteger(1000000, 5, 'COMMA')}
结果为：1,000,000
```

这些方法的第三个参数，要求传入特定的字符串来标识千位分隔符。

- POINT：使用“.”作为千位分隔符。
- COMMA：使用“,”作为分隔符。
- WHITESPACE：使用“ ”（空格）作为分隔符。
- NONE：不使用分隔符。
- DEFAULT：根据 Locale 对象来决定使用什么分隔符。

设置千位分隔符，要传入以上的字符串参数，否则在处理模板时，将会得到以下的异常信息：org.thymeleaf.exceptions.TemplateProcessingException: Unrecognized point format “”。

批处理方法：Thymeleaf 中的许方法都提供了批处理的方法，例如方法名为“abc”，则在很多情况下，会提供另外三个批处理方法：arrayAbc、listAbc、setAbc，这些方法分别用于处理数组、List 与 Set。读者在使用这些方法时，可以通过命名规则知道这些方法的作用。

6.3.2 小数格式化

小数格式化与整数格式化类似，也有四个处理方法，也可以传入单个数字、数组、List 与 Set。以下是这四个方法的描述。

- formatDecimal(num, intDig, decDig)：处理单个小数，第二个参数 intDig 表示最少的

整数位数，第三个参数 decDig 表示保留的小数位数。

- arrayFormatDecimal(numArray, intDig, decDig)：与 formatDecimal 类似，处理数组。
- listFormatDecimal(numList, intDig, decDig)：与 formatDecimal 类似，处理小数的 List。
- setFormatDecimal(numSet, intDig, decDig)：与 formatDecimal 类似，处理小数数组的 Set。

以下的代码片断，使用 formatDecimal 方法处理小数：

```
${#numbers.formatDecimal(10.992, 3, 2)}
```

这表示使用 formatDecimal 方法处理 10.992 这个小数，整数位最少为 3 位，小数位保留 2 位，该代码片断运行结果为：010.99。其他几个方法与整数格式化方法类似，传入什么类型的参数，方法将返回同样类型的结果，在此不再赘述。

与整数格式化类似，这四个方法也有重载方法，而且每个方法都有两个重载方法，以 formatDecimal 为例，它有以下两个重载方法。

- formatDecimal(number, intDig, decDig, decPoint)：比前面的方法多了一个 decPoint 参数，该参数表示使用哪种符号作为小数点，该参数的取值与千位分隔符一样，可使用 POINT、COMMA、WHITESPACE、NONE 与 DEFAULT 这五种值。
- formatDecimal(number, intDig, separator, decDig, decPoint)：比前一个方法多了 separator 参数，该参数表示使用哪种符号作为千位分隔符，同样可使用 POINT、COMMA 等五种值。

以下的代码片断，使用了以上两个重载的方法：

```
#numbers.formatDecimal(10.9929999, 3, 4, 'COMMA') // 结果 010,9930
#numbers.formatDecimal(1000000.9929999, 3, 'WHITESPACE', 3, 'COMMA') // 结果 1 000
000,993
```

第一行代码的运行结果为“010,9930”，小数点使用逗号；第二行代码运行结果为“1 000 000,993”，小数点使用逗号，千位分隔符使用空格。其他的三个小数格式化方法也有相同的重载方法，参数与 formatDecimal 类似。

6.3.3 货币格式化

#numbers 对象提供了 formatCurrency 等四个方法用来格式化数字，将它们转换为货币格式的字符串。如果 java.util.Locale 的 country（国家）属性设置为“CN”，则格式化“10.55”这个数字后，会得到“￥10.88”这样的字符串。实际上，formatCurrency 方法最终会使用如下的代码来格式化数字：

```
Locale locale = new Locale("zh", "CN", "");
java.text.NumberFormat format = NumberFormat.getCurrencyInstance(locale);
String result = format.format(10.88);
```

当然了，以上的代码片断定义了一个我们国家的 Locale。在模板中，使用以下方式调用 formatCurrency 方法：

```
<div th:text="${#numbers.formatCurrency(10.99)}"></div>
// 10.99 格式化后结果为￥10.99
```

除此之外，另外还有三个可以处理数组、List 与 Set 的方法，方法名分别为 arrayFormatCurrency、listFormatCurrency 与 setFormatCurrency。要处理什么类型的数值，传入相应的类型即可。

6.3.4 百分比格式化

百分比格式化与小数的格式化类似，调用 formatPercent 方法对数字进行百分比格式化，需要传入数字、整数最少位数、保留的小数位数三个参数，例如，以下代码片断使用 formatPercent 方法来格式化：

```
<div th:text="${#numbers.formatPercent(0.23456, 2, 3)}"></div>
```

要处理的值为“0.23456”，整数至少两位，小数保留三位，格式化后结果为 23.456%。与货币格式化类似，最终也会调用 java.text.NumberFormat 类进行格式化，当然也需要使用 Locale 类。除了这个方法外，还有另外三个用于处理数组、List 与 Set 的方法，在此不再赘述。有一点需要注意，官方文档指出，formatPercent 方法有一个重载方法，你只需要传入一个参数即可，但笔者在测试时并没有发现该方法。

6.3.5 使用 sequence 方法

调用#numbers 的 sequence 方法，会返回 Integer 数组。该方法可以设置开始值、结束值以及步长，它主要有以下两个重载的方法。

- sequence(from, to)：设置开始值与结束值，如果 from 比 to 大，则默认步长为 1，否则步长为–1。提示一下，如果开始值设置为 0，结束值设置为 3，则返回数组长度为 4。
- sequence(from, to, step)：设置开始值、结束值与步长。

代码清单 6-12 演示了如何使用 sequence 方法。

代码清单 6-12：codes\06\6.3\numbers\src\main\resources\sequence.html

```
<tr th:each="num : ${#numbers.sequence(1, 3)}"><td th:text="${num}"></td></tr>
```

由于 sequence 方法返回 Integer 数组，因此要使用 th:each 进行遍历。处理代码清单 6-12 的模板后，输出如下：

```
<tr><td>1</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
```

代码中的 th:each 与 sequence 配合使用，效果类似于 Java 的 for 循环：

```
for(int i = 1; i <= 3; i++) {…}
```

关于数字对象的讲解就到这里。Thymeleaf 几乎为每个功能都提供了重载的方法，并且很多方法都支持批量操作，可传入数组、List 与 Set 方法参数。这些设计其目的很简单，就是让我们更方便地使用#numbers 对象。

6.4 字符串对象

在模板中使用#strings 表示 org.thymeleaf.expression.Strings 对象，这一节介绍字符串对象的使用方法。

6.4.1 toString 与 length 方法

#strings 的 toString 方法，会使调用参数的 toString 方法返回字符串，length 方法则会返回参数的 toString 方法的 length，结果类型为 Integer。代码清单 6-13 测试了这两个方法的使用。

代码清单 6-13：codes\06\6.4\strings\src\main\java\org\crazyit\boot\c6\User.java
codes\06\6.4\strings\src\main\java\org\crazyit\boot\c6\TsLenTest.java
codes\06\6.4\strings\src\main\resources\tslen.html

```
User.java
public class User {

    private Integer id;
    private String name;

    ...省略其他代码

    @Override
    public String toString() {
        return "User [id=" + id + ", name=" + name + "]";
    }
}

TsLenTest.java
public class TsLenTest {

    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        Context ctx = new Context();
        ctx.setVariable("user", new User(1, "Angus"));
        String result = templateEngine.process("tslen.html", ctx);
        System.out.println(result);
    }
}

tslen.html
<div th:text="${#strings.toString(user)}"></div>
<div th:text="${#strings.length(user)}"></div>
```

代码清单 6-13 中的 User 类重写了 toString 方法，运行 TsLenTest 类，控制台输出如下：

```
<div>User [id=1, name=Angus]</div>
<div>23</div>
```

这两个方法的使用较为简单，要注意的细节是，官方文档介绍，只能给 length 方法传入字符串，但实际上可以传入 Object，最终会调用传入对象的 toString 方法，并返回 length。

这两个方法也有三个“批处理”方法，以 length 为例，有分别用来处理数组、List 与 Set 的 arrayLength、listLength 和 setLength 方法，返回的结果包含类型为 Integer 的元素。而 toString 方法则对应 arrayToString、listToString 与 setToString 三个方法，返回的结果包含类型为 String 的元素。

6.4.2 非空判断与默认值处理

#strings 提供了 isEmpty 与 defaultString 方法，用于进行字符串的非空判断与默认值处理，

以下是这两个方法的描述。

- isEmpty(Object s)：如果参数为 null，或者参数的 toString 方法返回的均是空格，则返回 true，否则返回 false。
- defaultString(Object s, Object defaultString)：该方法的判断逻辑类似于 isEmpty 方法，如果第一个参数是 null 或者是空格，则会返回默认字符串，如果默认字符串是 null，则返回字符串“null”。

这两个方法都有对应的批处理方法，isEmpty 的三个批处理方法为 arrayIsEmpty、listIsEmpty 与 setIsEmpty，返回的结果（数组或集合）包含类型为 Boolean 的元素；defaultString 方法同样有批处理方法 arrayDefaultString、listDefaultString 与 setDefaultString，返回的结果包含类型为 String 的元素。

6.4.3 包含判断

主要使用两个 contains 方法进行字符串包含判断。

- contains(Object obj, String s)：判断 obj 的 toString 方法返回值中，是否包含第二个参数，若包含则返回 true，否则返回 false。该方法也有三个批处理方法，arrayContains、listContains 与 setContains。
- containsIgnoreCase(Object obj, String s)：判断 obj 的 toString 方法的返回值是否包含第二个参数，这个方法在进行包含判断时会忽略大小写，该方法同样支持批处理。

除了以上两个字符串包含判断方法外，这里再介绍两个字符串开头与结尾的判断方法。

- startsWith(Object obj, String s)：判断 obj 的 toString 返回值是否以第二个参数开头，支持批处理。
- endsWith(Object obj, String s)：判断 obj 的 toString 返回值是否以第二个参数结尾，支持批处理。

这里简单讲下 Thymeleaf 对以上四个方法的实现。contains 方法最终会调用 java.lang.String 的 contains 方法进行包含判断；而 containsIgnoreCase 方法在调用 String 类的 contains 方法前，会调用 String 类的 toUpperCase 方法将两个字符串转换为大写形式；startsWith 与 endsWith 方法则分别调用 String 类的 startsWith 与 endsWith 方法，最后将结果转换为 Boolean 对象返回。

6.4.4 截取与替换

主要有以下几个字符串截取与替换的相关方法。

- indexOf(Object obj, String str)：先调用 obj 的 toString 方法得到字符串，再判断字符串 str 在该字符串中的索引，若不存在则返回-1。
- substring(Object obj, int start, end)：调用 toString 方法得到字符串，再进行截取，start 为开始索引，end 为结束索引，如果索引超过字符串长度，则会抛出以下异常信息：java.lang.StringIndexOutOfBoundsException: String index out of range。
- substringAfter(Object obj, String str)：从 obj 的 toString 方法返回值中截取字符串，从第二个参数 str 开始往后截取，不包括 str，如果字符串中没有 str，则返回空字符串。
- substringBefore(Object obj, String str)：从 obj 的 toString 方法返回值中截取字符串，

从第二个参数 str 开始往前截取，不包括 str，如果字符串中没有 str，则返回空字符串。

- replace(Object obj, String before, after)：调用 obj 的 toString 方法得到字符串，将字符串中的的 before（第一个参数）替换为 after（第二个参数），如果字符串中不包含 before，则返回原字符串。

以下的代码片断，是使用以上几个方法的例子：

```
<div th:text="${#strings.indexOf('abcde', 'z')}"></div>    // 结果为-1
<div th:text="${#strings.substring('abcde', 1, 3)}"></div>    // 结果为 bc
<div th:text="${#strings.substringAfter('abcde', 'a')}"></div>    // 结果为 bcde
<div th:text="${#strings.substringBefore('abcde', 'c')}"></div>    // 结果为 ab
<div th:text="${#strings.replace('abcde', 'c','1')} "></div>    // 结果为 ab1de
```

这几个方法都支持批量操作，例如 indexOf 有 arrayIndexOf、listIndexOf 与 setIndexOf 三个方法，它们返回的结果包含 Integer 类型的元素。

6.4.5 追加与拼接

使用#strings 的 prepend 与 append 方法可以实现字符串的追加。

- prepend(Object obj, String pre)：将 pre 字符串添加到 obj 前面，支持批处理。
- append(Object obj, String suf)：将 suf 字符串添加到 obj 后面，支持批处理。

以上两个方法的实现非常简单，使用“+”号将两个参数拼接并返回。除此之外，还有两个拼接方法，用于拼接字符串。

- concat(Object … values)：将传入的参数进行拼接，底层会遍历这些参数，并使用 StringBuilder 进行字符串拼接。
- concatReplaceNulls(String nullValue, Object … values)：同样将 values 参数进行拼接，如果参数是 null，则使用第一个参数（nullValue）替代。

append 与 concat 方法在实现上存在区别，append 方法最终会使用“+”号将字符串拼接并返回，而 concat 方法则会使用一个 StringBuilder 来保存全部的参数，最后返回 toString 方法。代码清单 6-14 示例了如何使用以上几个方法。

代码清单 6-14：codes\06\6.4\strings\src\main\resources\append.html

```
<div th:text="${#strings.prepend(3, 'e')}"></div>    // e3
<div th:text="${#strings.append('abcd', 'e')}"></div>    // abcde
<div th:text="${#strings.concat('abcd', 'e', 'b')}"></div>    // abcdeb
<div th:text="${#strings.concatReplaceNulls('**', '123', null, 'abc')}"></div>
// 123abc
```

注意最后的 concatReplaceNulls 方法，这个表达式的意思是，如果参数值中存在 null，则以“**”来代替 null 值，换言之，处理模板后结果应当是“123**abc”，但由于 OGNL 的 bug，在一般环境（OGNL 表达式）下，会输出“123abc”，该结果并不准确。将同样的表达式放到 Spring MVC 环境（SpringEL）下，输出结果则为“123**abc”，这是正确的结果，请注意这个细节。

注意：用于追加字符串的两个方法支持批处理，而两个拼接方法并没有批处理方法。

6.4.6 分割与连接

#string 支持将一定规则的字符串进行分割，并放到相应的数组或者集合中返回。主要有以下三个分割方法。

- arraySplit(Object str, String separator)：将 str 进行分割，separator 为分隔符，分割后的每个字符串都会被放到字符串数组中，例如字符串“a-b-c”，调用该方法后将得到一个数组，包含“a”“b”与“c”三个元素。
- listSplit(Object str, String separator)：与 arraySplit 类似，最后返回 List<String>。
- setSplit(Object str, String separator)：与 arraySplit 类似，最后返回 Set<String>。

有分割，也有连接，可以将数组或集合中的元素，按照一定的规则组合成一个字符串。有如下三个连接方法。

- arrayJoin (Object[] strs, String separator)：将 strs 中的全部元素连接成一个字符串，每个元素间使用 separator 作为分隔符。
- listJoin(List strs, String separator)：与 arrayJoin 方法类似，需要传入 List 参数。
- setJoin(Set strs, String separator)：与 arrayJoin 方法类似，需要传入 Set 参数。

代码清单 6-15 示例了如何使用 arraySplit 与 arrayJoin 方法。

代码清单 6-15：codes\06\6.4\strings\src\main\resources\split-join.html

```
    <tr th:each="str : ${#strings.arraySplit('a-b-c', '-')}"> // 结果为 String 数组
        <td th:text="${str}"></td>
    </tr>
    <tr th:text="${#strings.arrayJoin(new String[]{'a', 'b', 'c'}, '-')}"></tr>    //
结果为 a-b-c
```

分割与连接方法均没有批处理的方法。

6.4.7 大小写转换

#strings 提供了多个用于进行大小写转换的方法。

- toUpperCase(Object obj)：将参数全部转换为大写形式。
- toLowerCase(Object obj)：将参数全部转换为小写形式。
- capitalize(Object obj)：将参数的首字母转换为大写形式。
- unCapitalize(Object obj)：将参数的首字母转换为小写形式。
- capitalizeWords(Object obj)：将参数的每个单词的首字母转换为大写形式。
- capitalizeWords(Object obj, Object delimiters)：第二个参数为分隔符，该方法会按照分隔符，将各个单词的首字母转换为大写形式。

代码清单 6-16 示例了如何使用以上方法。

代码清单 6-16：codes\06\6.4\strings\src\main\resources\case.html

```
    <div th:text="${#strings.toUpperCase('疯狂 Spring Cloud')}"></div>    // 结果：疯狂
SPRING CLOUD
    <div th:text="${#strings.toLowerCase('疯狂 JAVA')}"></div>    // 结果：疯狂 java
```

```
    <div th:text="${#strings.capitalize('crazy')}"></div>    // 结果：Crazy
    <div th:text="${#strings.unCapitalize('Crazy')}"></div>    // 结果：crazy
    <div th:text="${#strings.capitalizeWords('疯狂 spring cloud')}"></div>    // 结果：
疯狂 Spring Cloud
    <div   th:text="${#strings.capitalizeWords('crazy-spring-cloud',   '-')}"></div>
// 结果：Crazy-Spring-Cloud
```

这 6 个方法都支持批处理，例如，toUpperCase，有 arrayToUpperCase、listToUpperCase 与 setToUpperCase 批处理方法，这些方法都会返回相应的数据类型的数据。

6.4.8 字符串编码与解码

在学习 Thymeleaf 的编码与解码方法前，我们先了解一下“unbescape”这个框架，这是一个专门用于编码和解码的 Java 库，它支持 HTML、XML、Java 等格式文本的编码和解码，Thymeleaf 就是使用 unbescape 进行编码与解码的。接下来，我们编写代码来使用 unbescape 的 API，请见代码清单 6-17。

代码清单 6-17：codes\06\6.4\strings\src\main\java\org\crazyit\boot\c6\EscapeTest.java

```
/*
 * 对 Java 语句进行编码和解码，级别 3，字母（大小写）与数字不会被编码，其他字符都会被编码
 */
String result = JavaEscape.escapeJava("System.out.println('你好');",
    JavaEscapeLevel.LEVEL_3_ALL_NON_ALPHANUMERIC);        ①
System.out.println(result);
// 解码
String unResult = JavaEscape.unescapeJava(result);
System.out.println(unResult);
/*
 * 对 HTML 进行编码和解码
 */
result = HtmlEscape.escapeHtml4Xml("<div>test</div>");            ②
System.out.println(result);
// 解码
unResult = HtmlEscape.unescapeHtml(result);
System.out.println(unResult);
/*
 * 对 JavaScript 进行编码和解码，默认级别为 2，对非 ASCII 等字符（例如中文）进行编码
 */
result = JavaScriptEscape
    .escapeJavaScript("var test = function() { alert('你好'); }");    ③
System.out.println(result);
// 解码
unResult = JavaScriptEscape.unescapeJavaScript(result);
System.out.println(unResult);
```

代码清单 6-17 中的①语句，使用 JavaEscape 对一小段 Java 语句进行编码，编码级别是 3，设置为该级别后，除了大小写字母、阿拉伯数字之外，其他的字符都会被编码。代码清单中的②语句，对 HTML 进行编码，一般会对 HTML 相关的字符（<、>、&、”、’）进行编码。代码清单中的③语句，对 JavaScript 语句进行编码，默认的级别为 2，对非 ASCII 字符进行编码，例如中文。在代码清单 6-17 中，在编码后都会执行相应的解码并输出。运行代码清单 6-17，控制台输出如下：

```
System\u002Eout\u002Eprintln\u0028\'\u4F60\u597D\'\u0029\u003B
```

```
System.out.println('你好');
&lt;div&gt;test&lt;/div&gt;
<div>test</div>
var test = function() { alert(\'\u4F60\u597D\'); }
var test = function() { alert('你好'); }
```

注意：加入 Thymeleaf 依赖后，可以直接使用 unbescape 库，不需要额外添加依赖。

下面，我们来看下#strings 对象提供的几个编码与解码的方法。

- escapeXml(Object str)：调用 HtmlEscape 类的 escapeHtml4Xml 方法对参数进行编码。
- escapeJava(Object str)：调用 JavaEscape 类的 escapeJava 方法对参数进行编码。
- escapeJavaScript(Object str)：调用 JavaScriptEscape 类的 escapeJavaScript 方法对参数进行编码。
- unescapeJava(Object str)：调用 JavaEscape 类的 unescapeJava 方法进行解码。
- unescapeJavaScript(Object str)：调用 JavaScriptEscape 类的 unescapeJavaScript 方法对参数进行解码。

注意：以上 5 个编码和解码方法都支持批处理。

6.4.9 其他处理

关于字符串处理还有几个方法，在此简单介绍一下。

- trim(Object obj)：删除参数前后的空白字符，会调用 String 的 trim 方法，支持批量处理。
- abbreviate(Object obj, int max)：将 obj 参数作“省略”处理，max 表示显示的字符位数，最小取值为 3。假设有字符串“AngusYoung”，max 取值为 5，则结果为“An...”。
- randomAlphanumeric(int count)：随机产生字母数字，参数 count 为产生的位数，例如若 count 取值为 5，则就会随机产生 5 位字符。随机取值的范围为：大写英文字母加上 0~9 阿拉伯数字。

关于字符串处理就讲解到这里，通过本节的学习，你应该能够在模板中使用#strings 对象来处理大部分的字符串问题了。这些方法的使用并不难，大家在使用过程中注意一些细节即可。

6.5 日期对象

在模板中可以使用#datas 或者#calendars 这两个对象来处理日期，实际上，这两个对象并没有太大的差别，它们的大部分方法名一致，参数一致，甚至它们底层的逻辑也是一致的。Thymeleaf 提供两个这样貌似重叠的对象，目的是为了方便我们使用。虽然两个对象大部分地方类似，但也存在一些区别，我们会在后面讲述它们间的区别。这里先讲解它们共同的方法。

6.5.1 格式化日期

主要有三个格式化日期的方法。

- format(Date date)：将参数的日期对象进行格式化，格式取决于 Locale 对象。
- formatISO(Date date)：将参数的日期对象格式化为 ISO8601 的格式。

仅供非商业用途或交流学习使用

➢ format(Date date, String pattern)：将参数按照 pattern 进行格式化。

代码清单 6-18 是一个使用以上三个格式化方法的示例。

代码清单 6-18：codes\06\6.5\dates\src\main\resources\format.html

```
<div th:text="${#dates.format(date)}"></div>
<div th:text="${#dates.formatISO(date)}"></div>
<div th:text="${#dates.format(date, 'yyyy-MM-dd HH:mm:ss')}"></div>
```

其中 date 参数是一个 java.util.Date 对象，需要将该对象设置到 Context 中：

```
Context ctx = new Context();
ctx.setVariable("date", new Date());
String result = templateEngine.process("format.html", ctx);
```

以上三个方法中的每一个方法都有三个批处理方法，以第一个 format 方法为例，它有 arrayFormat、listFormat 与 setFormat 批处理方法，换言之，在#dates 对象里面总共有 12 个（4 乘以 3）格式化日期的方法。

另外，我们可以使用#calendars 对象来代替#dates 对象，但使用#calendars 对象的格式化方法时，要传入 java.util.Calendar 对象参数。同样地，#calendars 对象也有 12 个格式化日期的方法，方法名与#dates 对象的一致（方法参数类型不同而已）。以下的代码片断在模板中使用#calendars 对象的 format 方法进行格式化：

```
<div th:text="${#calendars.format(cal)}"></div>
// 在 Java 类中设置 Calendar
ctx.setVariable("cal", Calendar.getInstance());
```

除了 format 方法外，#calendars 对象的其他方法也与#dates 对象的一致。#dates 与#calendars 两个对象共提供了 24 个格式化日期的方法。我们记住一个规则，使用#dates 对象就要传入 Date 对象，使用#calendars 对象就要传入 Calendar 对象。

注意：再次说明一下批处理方法的命名规则，假设方法名为“abc”，则三个批处理方法为“arrayAbc”、“listAbc”与“setAbc”。

6.5.2 获取日期字段

两个日期对象都提供了多个获取日期字符值的方法，这些方法在模板中的使用方法，请见代码清单 6-19。

代码清单 6-19：codes\06\6.5\dates\src\main\resources\prop.html

```
<div th:text="${#dates.day(date)}"></div>    // 获取当月的第几天，结果为 2（2 号）
<div th:text="${#dates.month(date)}"></div>    // 获取当前的月份，结果为 2
<div th:text="${#dates.monthName(date)}"></div>  // 获取当前月份的名称，结果为二月
<div th:text="${#dates.monthNameShort(date)}"></div> // 获取当前月份简称，结果为二月
<div th:text="${#dates.year(date)}"></div>    // 获取当前的年份，结果为 2018
<div th:text="${#dates.dayOfWeek(date)}"></div> // 获取当前是星期几，结果为 6（周五）
<div th:text="${#dates.dayOfWeekName(date)}"></div> // 获取当前星期几的名称，结果为星期五
<div th:text="${#dates. dayOfWeekNameShort (date)}"></div> // 获取当前星期几的简称，结果为星期五
<div th:text="${#dates.hour(date)}"></div>    // 获取当前的小时，结果为 23
<div th:text="${#dates.minute(date)}"></div>    // 获取当前分钟，结果为 2
<div th:text="${#dates.second(date)}"></div>    // 获取当前的秒，结果为 52
<div th:text="${#dates.millisecond(date)}"></div>    // 获取当前的毫秒，结果为 116
```

仅供非商业用途或交流学习使用

这些方法的使用比较简单，要注意几个实现细节。monthName 方法的实现会创建一个 SimpleDateFormat 的实例，格式为“MMMM”，再使用该实例去格式化参数的 Date；而 monthNameShort 方法使用的格式为“MMM”，请见以下代码片断：

```
    SimpleDateFormat sdf = new SimpleDateFormat("MMMM"); // monthName 方法使用
    SimpleDateFormat sdf = new SimpleDateFormat("MMM"); // monthNmonthNameShortame
方法使用
```

同样地，dayOfWeekName 与 dayOfWeekNameShort 方法有类似的实现，只不过使用的格式是“EEEE”与“EEE”。

以上的每一个方法都提供了三个批处理方法，以 month 为例，它提供了 arrayMonth、listMonth 与 setMonth 方法。代码清单 6-19 中使用了#dates 对象，可以将其替换为#calendar 对象，但所有方法的参数，都要传入 Calendar 对象。

6.5.3 创建日期

两个日期对象提供了多个创建日期（时间）对象的方法，以#calendars 对象为例，提供了下面这些方法。

- create(year, month, day)：根据参数的年、月、日，创建一个 Calendar 实例。
- create(year, month, day ,hour, minute)：同上，创建的 Calendar 实例会设置小时与分钟字段。
- create(year, month, day, hour, minute, second)：同上，会额外设置秒字段。
- create(year, month, day, hour, minute, second, millisecond)：同上，会额外设置毫秒字段。
- createNow()：创建当前时间的 Calendar 实例。
- createNowForTimeZone(Object tz)：与方法一类似，但要传入时区对象。
- createToday()：创建当前日期的 Calendar 实例，但不会设置小时、分钟、秒、毫秒这些值。
- createTodayForTimeZone(Object tz)：与前一方法类似，但要传入时区对象。

以上的方法都没有对应的批处理方法，在#dates 对象中同样存在这些方法，方法名与参数均一致，但是所有的方法，都会返回 java.util.Date 实例。以下的几个方法，只在#calendars 对象中存在。

- createForTimeZone(year,month,day,timeZone)：根据年、月、日与时区，创建一个 Calendar 实例。
- createForTimeZone(year,month,day,hour,minute,timeZone)：同上，可以指定小时与分钟字段。
- createForTimeZone(year,month,day,hour,minute,second,timeZone)：同上，可以指定秒字段。
- createForTimeZone(year,month,day,hour,minute,second,millisecond,timeZone)：同上，可以指定毫秒字段。

在实际应用中，我们都是在服务器后面进行时间的计算、处理，模板只负责展示结果，因

仅供非商业用途或交流学习使用

此建议重点掌握日期与时间的格式化方法。

6.6 数组与集合对象

Thymeleaf 提供了可以处理数组与集合的对象，这一节我们讲解几个对象的使用方法。

6.6.1 数组

数组对象#arrays 主要提供了以下方法。

- toArray(object)：将参数的 object 转换为数组，如果参数类型为数组，则直接返回 Object[]。如果参数类型是集合（实现了 Iterable 接口的类），则会转换为数组。
- toStringArray(object)：与 toArray 方法类似，参数可以是数组或集合，如果数组或集合中的元素不是字符串，则会抛出异常。除了此处的 toStringArray 方法外，还有 toIntegerArray、toLongArray、toDoubleArray、toFloatArray 与 toBooleanArray 方法，可以用来处理不同类型的数据。
- length(array)：得到数组的长度，参数也可以是集合。
- isEmpty(array)：判断参数的数组（或集合）是否为空。
- contains(array, element)：判断 array 中是否包含 element 元素。
- containsAll(array, elements)：判断 array 中是否包含全部的 elements 元素。

我们可以在模块中使用这样的表达式来定义数组：“{ele1, ele2...}”，也可以直接从 Context 中读取数组参数：“#numbers.length(arr)”。

#arrays 对象对应的 Java 类为 Arrays，在该类中都是一些普通的数组、集合操作逻辑，读者有兴趣可查看该类的源码。

6.6.2 List 与 Set

#lists 对象对应 Lists 类，主要有以下这几个方法。

- toList(object)：将对象转换为 List，如果参数是 List 类型，则直接返回；如果参数是数组，将调用 Arrays.asList 方法将其转换为 List；如果参数是 Iterable 类型，则新建一个 List 存放它们，最后返回该 List。
- sort(list)：将参数的 List 进行排序，需要先将集合转换为数组，再使用 Arrays.sort 方法进行排序。
- sort(list, comparator)：将参数的 List 进行排序，并传入自定义的 Comparator 实现类，其同样也会调用 Arrays 的 sort 方法进行排序。

除了以上几个方法外，还有 size、isEmpty、contains 等方法，这些方法与数组方法类似，一看方法名基本上就知道其作用。还有#sets 对象，访对象提供了一个 toSet 方法，它可以将参数转换为 Set，它的使用方法与#lists 对象的 toList 方法类似。由于 Set 是无序的，因此它没有排序方法。

6.6.3 Map

#maps 对象主要有以下几个 contains 方法。

仅供非商业用途或交流学习使用

- containsKey(map, key)：判断 Map 中是否存在键值为 key 的数据。
- containsAllKeys(map, keys)：判断 Map 的全部 key，是否都包含参数的 keys。
- containsValue(map, value)：判断 Map 中是否存在该 value 的数据。
- containsAllValues(map, values)：判断 Map 的全部 value，是否都包含参数的 values。

除了以上方法外，还有 size 与 isEmpty 方法，这两个方法可以计算 Map 的大小以及判断 Map 是否为空。

6.7 其他内置对象

本节介绍几个其他的 Thymeleaf 内置对象。

6.7.1 messages 对象

我们在本章的开头接触过#messages 对象，曾介绍过，“#{...}”（文本表达式）与“${#messages}”的效果一致。该对象主要有以下方法。

- msg(key)：提供外部文本的 key，也支持传入参数，还有 3 个重载方法。
- msgWithParams(key, objects)：与 msg 方法类似，可以传入参数数组，这些参数将会被替换到文本中。
- msgOrNull(key)：与 msg 方法类似，也有多个重载的方法。使用该方法，如果找不到对应的 key，则会返回空值，而使用 msg 方法，如果在外部文本中找不到对应的 key，则会返回类似“??key_zh_CN??”的字符串。
- msgOrNullWithParams(key, objects)：与前一个方法类似，可以传入参数数组。

在代码清单 6-20 中使用了 msg 与 msgWithParams 方法。

代码清单 6-20：codes\06\6.7\other-obj\src\main\resources\msg.html

```
    <span th:text="${#messages.msg('msgKey')}"></span>
    <span th:text="${#messages.msgWithParams('msgKey', new Object[]{'Angus',
'Paris'})}"></span>
```

msgWithParams 方法的第二个参数，直接在模板中创建 Object 数组。msg 与 msgOrNull 方法支持批量操作，例如，msg 方法有三个批处理方法：arrayMsg、listMsg 与 setMsg。

6.7.2 uris 对象

我们在前面学习了如何对字符串进行编码和解码，而#uris 对象提供了对 URI（或 URL）进行转码的功能，其主要有以下四个方法。

- escapePathSegment (uri)：对 URI 路径进行编码，大小写英文字母、阿拉伯数字以及部分字符不会被编码，这些字符包括“- . _ ~ ! $ & ' ( ) * + , ; = : @”。
- escapePath(uri)：会被编码的字符，在前一个方法（escapePathSegment）的基础上，多加一个“/”，
- escapeFragmentId(uri)：会被编码的字符，在前一个方法（escapePath）的基础上，多加一个“?”。

仅供非商业用途或交流学习使用

- escapeQueryParam(uri)：该方法主要对 URI 参数进行编码，除了英文字母与数字之外，还会对以下字符进行编码：“-._~!$'()*,;:@/?”。

大家只要大概了解以上的 4 个编码方法会处理哪些字符即可。有编码就会有解码，以上的 4 个方法，每一个都有 3 个派生出来的方法，以 escapePath 为例，它还有以下 3 个派生的方法。

- escapePath(uri, encoding)：使用特定的字符集进行编码，默认使用 UTF-8 进行编码。
- unescapePath(uri)：对 URI 进行解码。
- unescapePath(uri, encoding)：使用特定的字符集进行解码，默认使用 UTF-8。

其他 3 个编码方法，每一个也都有 3 个派生出来的方法，换言之，#uris 对象提供的处理 URI 的方法总共有 16 个（4 乘以 4）。

6.7.3 aggregates 对象

这个对象提供了两个工具方法，用来计算总数与平均数。

- sum(array)：计算数组中所有元素的总数，也可以传入集合参数。
- avg(array)：计算数组中所有元素的平均数，也可以传入集合。

加上重载的方法，aggregates 对象总共有 4 个方法可以使用，请见代码清单 6-21。

代码清单 6-21：codes\06\6.7\other-obj\src\main\resources\aggregates.html

```
<span th:text="${#aggregates.sum({10, 20, 30})}"></span>
<span th:text="${#aggregates.sum(#lists.toList({1, 2, 3}))}"></span>
<span th:text="${#aggregates.avg({10, 20, 30})}"></span>
<span th:text="${#aggregates.avg(#lists.toList({1, 2, 3}))}"></span>
```

执行代码清单 6-21，输出结果如下：

```
<span>60</span>
<span>6</span>
<span>20</span>
<span>2</span>
```

需要注意的是，如果传入非 Number 类型的参数，例如字符串，则会得到以下异常信息：java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Number。

另外，还有诸如#ids、#conversions 等对象，这些对象将会在本书相应的章节中讲解。

6.8 本章小结

本章的内容较多，主要讲解了 Thymeleaf 的大部分内置对象，读者学习完本章后，基本上可以掌握这些内置对象的使用方法，能在模板中应对一般的数据处理。实际上，我们在本章的学习过程，也是一个熟悉对象的过程，本章的知识看似松散，但你可以综合运用本章的知识，进行字符串、日期等多种数据的处理。另外，本章也可以作为 API 的文档，在后面章节的学习中，如果遇到内置对象使用的问题，可回头查阅本章内容。

仅供非商业用途或交流学习使用

CHAPTER

7

第7章 Thymeleaf常用语法

本章要点

- 运算符
- 条件判断
- 属性设置与数据迭代
- 变量优先级

仅供非商业用途或交流学习使用

我们在前面一章学习了 Thymeleaf 的内置对象，算是 Thymeleaf 语法的入门章节，这一章我们继续学习 Thymeleaf 的语法，包括运算符、变量等内容。

7.1 表达式语法之运算符

我们先学习表达式语法中的常量及运算符。

7.1.1 表达式常量

Thymeleaf 表达式中的“Literals”，有些人将其译作“字面量”，其属于表达式语法定义的常量，本书将其称作“表达式常量”。为简单起见，我们将在表达式中使用的 Literal 简称为“常量”。在 Thymeleaf 表达式中，主要可以定义以下几种常量。

- **字符串常量**：一般定义在单引号内，可以使用任何字符，如以下表达式定义了一个字符串常量。

```
<span th:utext="'Hello World'"></span>
```

- **数字常量**：如以下表达式。

```
<span th:text="2013"></span>  // 结果 2013
<span th:text="2013 + 2"></span>  // 结果 2015
```

- **布尔值常量**：可定义为 true 或者 false，例如以下表达式。

```
<div th:if="${result} == true">show text</div>
<div th:if="${result == false}">show text</div>
```

以上的两个表达式存在细微差别，将“== true”语句放到“${...}”语句外面，将会由 Thymeleaf 来处理，而如果将语句放到“${...}”里面，则会由 OGNL（一般环境）或者 SpringEL（Spring MVC 环境）来处理。

- **空值常量**：定义为 null，与布尔值常量类似。

```
<div th:if="${test} == null">show text</div>
<div th:if="${test == null}">show text</div>
```

在定义字符串常量时，也可以不使用单引号，例如以下代码：

```
<span th:text="Hello"></span>
```

这种写法，只能在双引号中出现以下几种字符：“大小写英文字母”、“阿拉伯数字”、“英文中括号（[]）”、“点（.）”、“连接号（-）”以及“下划线（_）”，例如字符串常量中没有使用单引号，而里面又含有逗号（或者空格）：

```
<span th:text="hello,world"></span>
```

执行以上的代码片断，将会得到类似于下面的异常信息：org.thymeleaf.exceptions.TemplateProcessingException: Could not parse as expression: "hello,world"。

提示：在处理字符串时，还是应该使用单引号（或“|”符号），以免产生混乱。

仅供非商业用途或交流学习使用

7.1.2 字符串拼接

实现字符串拼接的途径有多种，例如可以使用在前一章学习的#strings 对象，可调用其 append 或者 concat 方法来实现拼接。当然，也可以直接使用“+”号来拼接字符串，以下为几个拼接字符串的方法：

```
<div th:text="${#strings.append('abc', 'd')}"></div>
<div th:text="${#strings.concat('abc', 'd')}"></div>
<div th:text="${'abc' + 'd'}"></div>
<div th:text="'abc' + 'd'"></div>
<div th:text="abc + d"></div>
```

这几个拼接方法得到的结果都是“abcd”，后面三种方法都是使用“+”号来拼接。在这里再说明一下 append 与 concat 方法的区别，append 的底层也是使用“+”号进行字符串拼接，而 concat 则使用 StringBuilder 拼接。

使用加号拼接字符串，可能会涉及多个单引号内的字符串引加，字符串一多，很可能会产生混乱，鉴于此，我们可以使用“|”符号来代替单引号。与单引号类似，在“|”符号中间可出现任意字符，并且也可以将表达式放到里面，这一点是单引号不支持的，请见以下的代码片断：

```
<div th:text="|Hello, Angus|"></div>
<div th:text="|${name}|"></div>    // |中间支持使用表达式
<div th:text="'${name}'"></div>    // 单引号中不支持使用表达式
```

在“|”符号中间，除了可以使用“${...}”这种表达式外，还支持使用“#{...}”与“*{...}”表达式，关于“*{...}”表达式，会在后面章节中讲解。

“|”符号还可以与其他定义字符串的方式混合使用，以下的代码片断，混合使用了三种方式：

```
<div th:text="|Hello, Angus| + ' welcome to ' + ${company}"></div>
```

这里提示一下，如果在“|”中间使用布尔值、空值或者数字等，则效果与单引号一致，会被当作字符串进行处理，例如：

```
<div th:text="|2015| + |2|"></div>    // 结果为“20152”
<div th:text="|2015 + 2|"></div>    // 结果为“2015 + 2”
```

另外，如果在表达式“${...}”中间使用“|”符号，则会抛出异常，使用单引号则不会。

7.1.3 算术运算符

Thymeleaf 表达式支持几个常用的算术运算符：加（+）、减（-）、乘（*）、除（/）与求余（%），这几个算术运算符，可以在 OGNL 表达式内使用，也可以在 OGNL 表达式外使用，请见以下的代码片断：

```
<div th:text="${3 + 2}"></div>  等价于  <div th:text="3 + 2"></div>
<div th:text="${5 - 2}"></div>  等价于  <div th:text="5 - 2"></div>
<div th:text="${5 * 2}"></div>  等价于  <div th:text="5 * 2"></div>
<div th:text="${5.0 / 2}"></div>  等价于  <div th:text="5.0 / 2"></div>
<div th:text="${5 % 2}"></div>  等价于  <div th:text="5 % 2"></div>
```

仅供非商业用途或交流学习使用

其中除（/）与求余（%）这两个运算符，可以使用它们的别名 div（/）与 mod（%），但这两个别名，不能在 OGNL 表达式中使用，请见代码片断：

```
<div th:text="4 div 2"></div>    // 结果为2
<div th:text="5 mod 2"></div>    // 结果为1
<div th:text="${5 mod 2}"></div>    // 抛出异常
```

以上代码片断的第 3 行，在表达式中使用了 mod 别名，这在 OGNL 环境中，将会抛出异常，信息为：org.thymeleaf.exceptions.TemplateProcessingException: Exception evaluating OGNL expression: "5 mod 2"。

除（/）与求余（/）的别名，不能在 OGNL 表达式中使用，但在 SpringEL 表达式中则可以正常使用，请读者注意这个细节，建议尽量不要使用别名。

7.1.4 关系运算符

Thymeleaf 支持以下几个关系运算符：>、<、>=、<=、==与!=，这些关系运算符较为常见，使用比较简单，在此不过多赘述。这里讲一下这些运算符的特别之处，其中要注意的是大于号（>）和小于号（<），为了避免这两个符号与 XML 文档中的“< >”混淆，我们可以在模板中使用“<”与“>”来表示它们，这两个别名，同样支持在 OGNL 中使用，请见以下的代码片断：

```
<div th:text="2 &gt; 1"></div>  等价于  <div th:text="2 > 1"></div>
<div th:text="${2 &gt; 1}"></div>  等价于  <div th:text="${2 > 1}"></div>
```

以上是两个特别的运算符的别名，这 6 个关系运算符，每一个都有别名，分别为：gt（>）、lt（<）、ge（>=）、le（<=）、eq（==）与 neq/ne（!=），其中不等于（!=）有两个别名，请见以下代码片断：

```
<div th:text="1 gt 1"></div>  等价于  <div th:text="1 > 1"></div>
<div th:text="1 lt 1"></div>  等价于  <div th:text="1 < 1"></div>
<div th:text="1 ge 1"></div>  等价于  <div th:text="1 >= 1"></div>
<div th:text="1 le 1"></div>  等价于  <div th:text="1 <= 1"></div>
<div th:text="1 eq 1"></div>  等价于  <div th:text="1 == 1"></div>
<div th:text="1 ne 1"></div>  等价于  <div th:text="1 ! = 1"></div>
<div th:text="1 neq 1"></div>  等价于  <div th:text="1 != 1"></div>    // 不得在SpringEL中使用
```

以上的关系运算符别名，都没有在 OGNL 中使用，其中有 3 个运算符在 OGNL 中的别名与上面的不同，请见以下代码：

```
<div th:text="${1 gte 1}"></div>  等价于  <div th:text="${1 >= 1}"></div>
<div th:text="${1 lte 1}"></div>  等价于  <div th:text="${1 <= 1}"></div>
<div th:text="${1 neq 1}"></div>  等价于  <div th:text="${1 != 1}"></div>
```

在 OGNL 中，大于等于（>=）使用“gte”别名，小于等于（<=）使用“lte”别名，不等于（!=）使用且只能使用“neq”别名。

相对于 OGNL，在 SpringEL 中，这些运算符的别名则更接近于 Thymeleaf 中的别名，除了不等于（!=）的 neq 别名不能在 SpringEL 中使用外，其他别名均与 Thymeleaf 中的一致，即 SpringEL 使用的别名为：gt（>）、lt（<）、ge（>=）、le（<=）、eq（==）与 ne（!=）。如果

仅供非商业用途或交流学习使用

在 SpringEL 中使用了 neq 别名，则会得到以下异常：org.springframework.expression.spel.SpelParseException: EL1041E: After parsing a valid expression, there is still more data in the expression: 'neq'。

7.1.5 条件运算符

条件运算符表达式为：(condition) ? then : else，当条件 condition 成立时返回 then，否则返回 else，也可以称其为三目运算符，请见以下的代码片断：

```
<div th:text="${1 != 1} ? |Angus| : |Paris|"></div>
```

当表达式“${1 != 1}”成立时返回“Angus”，否则返回“Paris”。三目运算符还有另外一种写法，即可以省略 then 语句：

```
<div th:text="${1 != 1} ? |Angus|"></div>
```

表达式成立则返回字符串，不成立则直接返回 null。还有另外一种写法，也可以算作三目运算符，叫作默认值表达式：“(value) ?: (defaultValue)”，当存在某个值时，直接返回该值，否则返回默认值，对于 value 表达式，只需要提供一个值即可，并不需要提供条件判断。以下两行代码，效果一致：

```
<div th:text="${name} ?: |Angus|"></div>
<div th:text="${value != null} ? ${value} : |Angus|"></div>
```

再次提示，使用默认值表达式，不需要判断值是否为 null。

7.1.6 无操作符

使用“_”符号表示无操作，当一个值不存在时，可以使用该符号指定表达式不进行任何操作，例如页面设计同事提供了以下的原型文件：

```
用户：<span>小明</span>
```

程序员在开发模板时，可能会使用以下条件语句：

```
用户：<span th:text="${user.name} ?: '小明'"></span>
```

可使用以下的语句替换上面的语句：

```
用户：<span th:text="${user.name} ?: _">小明</span>
```

使用无操作符后，当 user.name 不存在时，表达式不进行任何操作，直接返回原型的内容。这两种写法的结果虽然一致，但根据上面的语句可知，使用无操作符，对原型界面的破坏是最小的。

7.2 表达式语法进阶

这一节我们学习 Thymeleaf 中的数据转换功能以及表达式的预处理功能，这两个功能涉及使用“${{...}}”与“__${...}__”这两个表达式。

7.2.1 数据转换

我们可以使用“${{...}}”这样的表达式来进行数据转换，Thymeleaf 将会使用配置好的

仅供非商业用途或交流学习使用

数据转换类，来实现转换过程，默认只会调用需要转换对象的 toString 方法得到结果。例如在 Context 中设置了一个 User 对象，请见代码清单 7-1。

代码清单 7-1：codes\07\7.2\exp-imp\src\main\java\org\crazyit\boot\c7\ConverTest.java

```
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        Context ctx = new Context();
        ctx.setVariable("user", new User("Angus", 33));
        String result = templateEngine.process("conver.html", ctx);
        System.out.println(result);
```

模板 conver.html 的内容如下：

```
<div th:text="${{user}}"></div>
<div th:text="${user}"></div>
```

默认的转换类仅仅调用 toString 方法，因此本例模板中的两行代码的结果一致，“${{...}}”表达式并没有产生作用。显然，默认的实现并不能满足所有需求，有时需要考虑使用自定义数据转换类。

7.2.2 自定义数据转换类

假设现在有一个需求，即在模板中显示用户对象时，要隐藏用户的年龄，为此，可以编写一个自定义的数据转换类，请见代码清单 7-2。

代码清单 7-2：codes\07\7.2\exp-imp\src\main\java\org\crazyit\boot\c7\MyConverService.java

```
public class MyConverService implements IStandardConversionService {

    public String convert(IExpressionContext context, Object object,
            Class targetClass) {
        // 只处理用户对象
        if (object instanceof User) {
            // 强制转换为 User 对象
            User u = (User) object;
            // 只返回名称
            return "name: " + u.getName();
        }
        return object.toString();
    }
}
```

在代码清单 7-2 中定义了一个 IStandardConversionService 接口的实现类，其在进行数据转换时，只会处理 User 对象，简单返回 name 字符串。接下来将这个对象注册到模板引擎中，请见代码清单 7-3。

代码清单 7-3：codes\07\7.2\exp-imp\src\main\java\org\crazyit\boot\c7\MyConverTest.java

```
TemplateEngine templateEngine = new TemplateEngine();
templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
// 创建自定义转换类
IStandardConversionService customConversionService = new MyConverService();
// 设置自定义转换类
StandardDialect dialect = new StandardDialect();
dialect.setConversionService(customConversionService);
templateEngine.setDialect(dialect);
```

仅供非商业用途或交流学习使用


```
// 设置用户参数
Context ctx = new Context();
ctx.setVariable("user", new User("Angus", 33));
String result = templateEngine.process("my-conver.html", ctx);
System.out.println(result);
```

在模板中使用“${{...}}”表达式来输出 User 对象：

```
<div th:text="${{user}}"></div>
<div th:text="${user}"></div>
```

虽然本例的模板与 7.2.1 节中的一致，但是输出结果完全不同。运行代码清单 7-3，输出结果如下：

```
<div>name: Angus</div>
<div>name: Angus, age: 33</div>
```

根据结果可知，User 对象的 age 属性没有被输出，这说明我们自定义的数据转换类生效了。

除了使用“${{...}}”这样的语法外，使用#conversions 对象也可以收到同样的效果，例如，以下的模板使用了#conversions 对象：

```
<div th:text="${#conversions.convert(user, 'java.lang.String')}"></div>
```

在该模板中使用了#conversions 对象的 covert 方法，该方法会将 user 转换为 java.lang.String。在转换类的 convert 方法中，第三个参数（targetClass）会得到 String 的 class，由于本例中的 convert 方法并没有使用 targetClass，因此传入什么类型的数据都不会影响转换结果。

7.2.3 Spring Boot 数据转换

Spring 框架里面已经包含了数据转换（格式化）的机制，在 Spring Boot+Thymeleaf 这样的框架组合中，我们同样可以在模板中使用数据转换功能。Spring 默认提供了很多数据转换类，基本上可以满足我们的需求，例如有日期转换为字符串、字符串转换为货币等转换类，我们甚至可以很轻松地提供自定义转换类。下面，我们将在 Spring Boot+Thymeleaf 的环境中，使用自定义的数据转换类。

新建一个 Spring Boot 项目，并将 Thymeleaf 整合到项目中，整合方式及配置方法请见第 5 章。新建两个格式化类，分别用来处理 BigDecimal 与 Date 类型数据，请见代码清单 7-4。

代码清单 7-4：codes\07\7.2\exp-boot\src\main\java\org\crazyit\boot\c7\BigDecimalFormatter.java
codes\07\7.2\exp-boot\src\main\java\org\crazyit\boot\c7\DateFormatter.java

```
public class BigDecimalFormatter implements Formatter<BigDecimal> {

    @Override
    public String print(BigDecimal object, Locale locale) {
        return "$" + object.toString() + "$";
    }

    @Override
    public BigDecimal parse(String text, Locale locale) throws ParseException {
        return null;
    }
}
```

```
public class DateFormatter implements Formatter<Date> {

    @Override
    public String print(Date object, Locale locale) {
        try {
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH时mm分ss秒");
            return sdf.format(object);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return object.toString();
    }

    @Override
    public Date parse(String text, Locale locale) throws ParseException {
        return null;
    }
}
```

在处理 BigDecimal 数据时，会在数据的前后加上“$”符号；在处理 Date 数据时，会将日期时间格式化为“yyyy-MM-dd HH 时 mm 分 ss 秒”形式。在实现 Formatter 类时要实现 print 与 parse 方法，print 方法用来将对象转换为字符串，而 parse 方法则用来将字符串转换为对象，由于这里我们只需要将对象显示到模板中，因此实现 print 方法即可。

接下来，将这两个格式化类注册到 Spring 容器中。新建一个配置类，请见代码清单 7-5。

代码清单 7-5：codes\07\7.2\exp-boot\src\main\java\org\crazyit\boot\c7\MyConfig.java

```
@Configuration
public class MyConfig {

    @Bean
    public BigDecimalFormatter bigDecimalFormatter() {
        return new BigDecimalFormatter();
    }

    @Bean
    public DateFormatter dateFormatter() {
        return new DateFormatter();
    }
}
```

本例的启动类与控制器为同一个类。新建控制器方法，设置相应的数据，请见代码清单 7-6。

代码清单 7-6：codes\07\7.2\exp-boot\src\main\java\org\crazyit\boot\c7\ThyApp.java

```
@SpringBootApplication
@Controller
public class ThyApp {

    public static void main(String[] args) {
        SpringApplication.run(ThyApp.class, args);
    }

    @RequestMapping("/test")
    public String test(Model model) {
        model.addAttribute("money", new BigDecimal(1000.00));
```

```
        model.addAttribute("date", new Date());
        return "test";
    }
}
```

在模板文件中，直接使用 th:text 输出两个参数：

```
<div th:text="${{money}}"></div>
<div th:text="${{date}}"></div>
```

运行启动类，在浏览器中访问：http://localhost:8080/test，浏览器输出如下：

```
$1000$
2018-02-09 17时01分54秒
```

根据结果可知，我们提供的两个格式化类已经生效。这里再讲一下实现顺序。先实现自定义的 Formatter 类，并根据具体的业务来实现数据转换逻辑，再将自定义的 Formatter 类注册到 Spring 容器中，最后在模板中使用“${{...}}”这样的语句，就可以根据相应的数据类型，使用不同的 Formatter 了。

有读者可能会问，如果存在两个类型相同的 Formatter，那么最终会使用哪个 Formatter 呢？根据笔者测试，在配置类中，在后面配置的 bean 将会生效，请见以下代码片断：

```
@Configuration
public class MyConfig {

    @Bean
    public BigDecimalFormatterB bigDecimalFormatteB() {
        return new BigDecimalFormatterB();
    }

    @Bean
    public BigDecimalFormatter bigDecimalFormatter() {
        return new BigDecimalFormatter();
    }
}
```

该代码中的 BigDecimalFormatterB 与 BigDecimalFormatter 类都用来格式化 BigDecimal 类型的数据，而最终生效的是 BigDecimalFormatter（下面一个 bean），因为在配置类中，BigDecimalFormatter 被定义在后面，读者在使用时注意这个细节。

7.2.4 表达式预处理

在了解表达式预处理功能前，我们先看一个小例子。假设在模板中需要读取外部文本，你可以使用文本表达式（“#{key}”），也可以使用#messages 对象：“${#messages.msg(key)}”。假如 key 是动态的，那么以下的表达式，将无法输出我们想要的结果：

```
<div th:text="#{user.key}"></div>
```

以上代码会根据 User 的 key 属性来决定读取哪个 properties 属性，但即使设置了正确的 key，在执行以上的模板后，会输出以下的结果：

```
<div>??user.key_zh_CN??</div>
```

该结果并不是我们想要的，这个结果明显是由于“#{...}”表达式先于 user.key 执行，而

导致无法读取相应的 key 造成的。为了解决这个问题，可以使用表达式的预处理功能，让 user.key 先执行，最后再执行“#{...}”表达式。非常简单，只要对模板作相应的修改即可：

```
<div th:text="#{__${user.key}__}"></div>
```

只要使用“__${...}__”这样的表达式，将要处理的表达式包含起来即可。以上例子中的 user.key 会被优先处理，最后“#{...}”表达式会读取 user.key 的值，来寻找相应的外部文本。

7.2.5 表达式调用工具方法

在表达式中，可以使用“@全限定类名@方法名(参数)”这样的格式来调用 Java 类的方法。新建一个 Java 类，请见代码清单 7-7。

代码清单 7-7：codes\07\7.2\exp-imp\src\main\java\org\crazyit\boot\c7\MyTranslator.java

```
public class MyTranslator {

    public static String testTrans(String sourceName) {
        System.out.println("要转换的数据：" + sourceName);
        return sourceName.toUpperCase();
    }
}
```

MyTranslator 类提供了一个静态方法，它会将参数的字符串转换为大写并返回。那么在模板中，就可以使用以下的方式来调用该方法：

```
<div th:text="|Hello, | + ${@org.crazyit.boot.c7.MyTranslator@testTrans
('Angus')}"></div>
```

输入的参数为“Angus”，执行模板后，结果如下：

```
<div>Hello, ANGUS</div>
```

在实际应用中，参数很有可能是动态的，对于这些参数，可以结合使用预处理功能。修改模板内容，读取动态参数：

```
<div th:text="|Hello, | + ${@org.crazyit.boot.c7.MyTranslator@testTrans
('__${user.key}__')}"></div>
```

在模板中会预处理“${user.key}”表达式，然后调用工具方法将该值转换为大写，最终输出结果如下：

```
<div>Hello, KEY1</div>
```

本例调用了 Java 类的方法来处理参数，也使用了预处理功能。可根据这个例子，深入地了解 Thymeleaf 的预处理功能，实际上就是优先执行某些表达式，以达到期望的效果。运行 codes\07\7.2\exp-imp\src\main\java\org\crazyit\boot\c7\ProcessTest.java，控制台输出如下：

```
要转换的数据：Angus
要转换的数据：key1
<div>This is key 1</div>
<div>Hello, ANGUS</div>
<div>Hello, KEY1</div>
```

7.3 属性设置

这一节介绍如何使用 Thymeleaf 的属性来设置 HTML 属性。

7.3.1 使用 th:attr

在前面章节中，我们使用 th:text 属性来输出普通文本。在模板文件中，可以为节点添加 th:attr 属性，使用该属性来指定 HTML 节点的属性。在实际开发中，界面设计同事可能会给出一份这样的原型文件：

```
<form>
    <input type="text" name="userName" value="小明"/>
    <input type="submit" value="提交"/>
</form>
```

表单中的“提交”按钮，在项目中可能需要对其进行国际化。为了不破坏原来页面的结构，可以使用 th:attr 属性来完成，修改后的模板文件内容如下：

```
<form>
    <input type="text" name="userName" value="小明"/>
    <input type="submit" value="提交" th:attr="value=#{myKey}"/>
</form>
```

使用 th:attr 属性，修改原来 HTML 节点的 value 属性，该 value 属性的值由 myKey 变量决定。使用 th:attr 属性不仅可以设置任何 HTML 属性，也可以同时指定多个属性：

```
<form th:attr="method='post', name=#{'formName'}, action=@{/user/save}">
    <input type="text" name="userName" value="小明"/>
    <input type="submit" value="提交" th:attr="value=#{myKey},class='button'"/>
</form>
```

在以上的 HTML 表单中，不仅在 th:attr 中定义了常量字符串（method='post'），还使用链接表达式（“@{...}”）设置了 URL。处理以上的模板文件后，输出如下：

```
<form method="post" name="myForm" action="/user/save">
    <input type="text" name="userName" value="小明"/>
    <input type="submit" value="Submit In English" class="button"/>
</form>
```

7.3.2 设置 HTML 节点属性

使用 th:attr 属性可以设置任何 HTML 节点的属性，但设置多个属性时，需要使用字符串将它们拼接起来，这显得不那么优雅，然而我们可以使用具体的 Thymeleaf 属性来设置，例如以下的语句：

```
th:attr="value='my value'"
```

可以被替换为：

```
th:value="my value"
```

每一个 HTML 属性都有对应的 Thymeleaf 属性，例如前一小节中 form 表单的几个属性，可以使用以下的代码片断代替：

```
<form th:method="post" th:name="formName" th:action="@{~/user/save}">
    <input type="text" name="userName" value="小明"/>
    <input type="submit" value="提交" th:value="#{myKey}" th:class="button"/>
</form>
```

在HTML中，有一些布尔型的属性，例如checkbox的checked属性，对于这种属性，浏览器支持以下两种写法：

```
<input type="checkbox" checked/>
<input type="checkbox" checked="checked"/>
```

在Thymeleaf模板中，只需要写成th:checked="true/false"的形式，处理后结果为checked="checked"。除了checked属性外，还有readonly、required、disabled等属性，都可以使用th属性来处理。

7.3.3 属性值拼接

Thymeleaf提供了属性值拼接的功能，可以在HTML属性的前面或者后面加入数据。以下的模板，分别在HTML属性的前面和后面加入内容：

```
<div id="imgDiv" th:attrappend="id='-dataId'"></div>    // 在id属性后面加入-dataId
字符串
<div id="imgDiv" th:attrprepend="id='data-'"></div>    // 在id属性前面加入data-字
符串
```

以上两个th属性，可以操作各个HTML属性。在实际应用中，比较常见的可能就是对HTML样式的操作。不同的数据，可能需要显示为不同的样式，为此，Thymeleaf提供了两个特别的属性，允许我们为class、style两个属性添加内容，并设置两个参数到Context中，请见以下Java代码：

```
Context ctx = new Context();
ctx.setVariable("appendClass", "red");
ctx.setVariable("appendStyle", "color: red;");
```

在模板文件中，向原有的class、style属性添加样式：

```
<div style="text-align: center;" th:styleappend="${appendStyle}"></div>
<div class="textClass" th:classappend="${appendClass}"></div>
```

模板的第一行，原来div的style为“text-align: center;”，表示文本居中，再使用th:styleappend属性加入“color: red;”，表示字体要设置为红色。第二行，原来的class值为“textClass”，再使用th:classappend属性加入“red”。执行模板文件，输出结果如下：

```
<div style="text-align: center; color: red"></div>
<div class="textClass red"></div>
```

我们可以选择使用th:attrappend、th:classappend或者th:styleappend来动态修改样式，这样做不会破坏原有的界面。

7.3.4 HTML5属性支持

HTML5支持使用自定义的节点属性，你可以在节点中加入类似“data-my-attribute”的属性，这些属性需要以“data-”作为前缀。例如以下的HTML代码：

```
<input id="myText" type="text" data-person-name="angus" data-age="red"/>
<script>
    var text = document.getElementById("myText");
    alert(text.dataset.personName);
    alert(text.dataset.age);
</script>
```

在这段 HTML 中，input 节点有 data-person-name 与 data-age 两个自定义属性。在 JavaScript 中，可以直接使用 text.dataset.personName 来获取属性值。Thymeleaf 同样支持自定义属性，例如可以使用“data-th-text”来代替 th:text 属性。在以下的代码片断中，使用了 HTML5 的自定义属性：

```
<table>
    <tr data-th-each="user : ${users}">
        <td data-th-text="${user.id}">...</td>
    </tr>
</table>
```

在该模板中，使用“data-th-each”来代替 th:each，使用“data-th-text”来代替 th:text。使用这种自定义的属性，更加符合 W3C 的规范，而且使用这种方式，并不意味着要替换掉原来的 th 语法，而是使使用者多一个语法上的选择。

7.4 条件判断与数据迭代

条件判断与数据迭代是最重要的语法，这一节我们讲解使用它们的方法。

7.4.1 条件判断

假设在页面开发中，需要根据用户的性别字段（int 类型），来显示“男”或者“女”字符串。以前使用 JSTL 或者其他 JSP 标签，可以这样来实现（例如使用 JSTL）：

```
<c:if test="${user.gender == 0}"><span>男</span></c:if>
<c:if test="${user.gender == 1}"><span>女</span></c:if>
```

而在 Thymeleaf 模板中，就需要使用 th:if 属性来实现：

```
<span th:text="男" th:if="${user.gender == 0}"></span>
<span th:text="女" th:if="${user.gender == 1}"></span>
```

把判断的逻辑直接写到 span 节点里面。在上面例子中，th:if 判断“${...}”表达式是否成立，该表达式的结果为 boolean 类型，除了 boolean 类型外，还支持数字（number）、字符（character）、字符串（String）以及其他类型。

如果表达式的结果是一个数字且不是 0，则 if 语句成立；如果表达式的结果是一个字符且不是 0，则 if 语句也成立；如果表达式的结果是一个字符串且不是“false”、“off”或者“no”，则 if 语句也成立；如果表达式的结果不是 4 种基本类型之一，则 if 语句也成立。以下为几种类型的语句：

```
// 以下 if 语句成立
<span th:text="非 0" th:if="1"></span>    // 数字类型，如果非 0，则成立
<span th:text="非 0" th:if="'1'"></span>    // char 类型，如果非 0，则成立
<span th:text="非 false、off、no 字符串" th:if="'true'"></span>    // 如果不是 false、
```

```
off、no 字符串，也成立
    <span th:text="非基本类型" th:if="${user}"></span> // 非基本类型，其他类型条件也成立

    // 以下 if 语句不成立
    <span th:text="false 字符串" th:if="'false'"></span>
    <span th:text="off 字符串" th:if="'off'"></span>
    <span th:text="no 字符串" th:if="'no'"></span>
```

执行以上模板，前 4 个 span 节点均会输出，后面 3 个 span 节点，由于 if 条件并不成立，因此不会输出。th:if 属性还有一个相反的属性 th:unless，它的使用方法与 th:if 属性相反，在此不再赘述。

关于表达式的结果类型，笔者建议还是使用 boolean 类型，这样才比较符合逻辑，不易发生混乱。

7.4.2 switch case 语法

该语法类似于 Java 的 switch case 语法。我们下面使用一个 HTML 中的下拉框来测试 switch case 语法，请见以下的代码片断：

```
<select th:switch="${user.gender}">
    <option th:case="0">男</option>
    <option th:case="1">女</option>
</selecdt>
```

使用 th:switch 来获取性别属性（int 类型），再使用 th:case 属性，根据不同的属性值显示不同的 option 节点。如果例子中 user.gender 的值不是 0 也不是 1，则需要增加一个“其他”选项，我们来修改以上的代码片断：

```
<select th:switch="${user.gender}">
    <option th:case="0">男</option>
    <option th:case="1">女</option>
    <option th:case="*">其他</option>
</selecdt>
```

使用“th:case="*"”来表示默认值，这样的语法类似于 Java 语言中的 default 语句。需要注意的是，如果第一个 th:case 的结果为 true，那么其他所有的 th:case 项都会被设置为 false，即使它们的结果也是 true。

7.4.3 使用 th:each 属性

前面章节的很多例子都使用过 th:each 属性。下面的代码片断对一个 List 进行迭代：

```
<table>
    <tr th:each="user : ${users}">
        <td th:text="${user.name}"></td>
    </tr>
</table>
```

该代码片断对“${users}”变量进行迭代，每一次迭代都会将当前节点设置到名称为“user”的变量中。“${users}”被称为“可迭代变量”，而 user 则属于“迭代变量”。迭代变量除了节点之外，还包括迭代过程中产生的变量，例如迭代索引、数据大小等，这将在后面讲解。

7.4.4 常见迭代类型

th:each 属性支持的迭代类型有很多，最常见的就是数组、List、Set 和 Map，其中数组、List、Set 的迭代方法类似，而迭代 Map 时，会得到一个 java.util.Map.Entry 对象。下面，我们对 List 与 Map 数据进行迭代，并将 List 与 Map 数据放到 Context 中，请见代码清单 7-8。

代码清单 7-8：codes\07\7.4\iter-test\src\main\java\org\crazyit\boot\c7\IterTest.java

```
TemplateEngine templateEngine = new TemplateEngine();
templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
Context ctx = new Context();
// 将 List 放到 Context 中
List<User> users = new ArrayList<User>();
users.add(new User(1, "Angus"));
users.add(new User(2, "Paris"));
ctx.setVariable("users", users);
// 将 Map 放到 Context 中
Map<String, User> userMap = new HashMap<String, User>();
userMap.put("a", new User(3, "Angus Map"));
userMap.put("b", new User(4, "Paris Map"));
ctx.setVariable("userMap", userMap);
String result = templateEngine.process("iter.html", ctx);
System.out.println(result);
```

在模板文件中迭代 List 与 Map，请见代码清单 7-9。

代码清单 7-9：codes\07\7.4\iter-test\src\main\resources\iter.html

```
<table>
   <tr th:each="user : ${users}">
      <td th:text="${user.name}"></td>
   </tr>
</table>
<table>
   <tr th:each="map : ${userMap}">
      <td th:text="${map.key} + |---| + ${map.value.name}"></td>
   </tr>
</table>
```

在迭代 List 时，放入什么就出来什么。而针对 Map，出来的是一个 Entry，可以直接调用 Entry 的 key 或者 value 来取得键值。

7.4.5 迭代状态对象

在迭代过程中，我们需要获取一些迭代状态的变量，并通过这些变量来处理模板节点，例如需要获取当前迭代的索引、数据集合的大小等变量，则在迭代时，我们可以使用以下内置的变量。

- index：迭代索引，从 0 开始。
- count：迭代索引，从 1 开始。
- size：数据集合的大小。
- current：当前的迭代节点。

- first：当前是否第一次迭代。
- last：当前是否最后一次迭代。
- even 与 odd：两个值均为 boolean 值，如果 odd 为 true，则表示当前迭代次数为奇数，如果 even 为 true，则表示当前迭代次数为偶数。例如第一次迭代，odd 为 true，even 为 false。需要注意，迭代次数并非索引，第一次对集合进行迭代，index 索引值为 0。

编写模板，在迭代过程中获取这些变量，请见代码清单 7-10。

代码清单 7-10：codes\07\7.4\iter-test\src\main\resources\status.html

```
<table>
   <tr th:each="data,iterState : ${#numbers.sequence(0, 2)}">
      <td th:text="|从 0 开始的索引：| + ${iterState.index}"></td>
      <td th:text="|从 1 开始的索引：| + ${iterState.count}"></td>
      <td th:text="|数据集合大小：| + ${iterState.size}"></td>
      <td th:text="|当前节点：| + ${iterState.current}"></td>
      <td th:text="|是否第一次迭代：| + ${iterState.first}"></td>
      <td th:text="|是否最后一次迭代：| + ${iterState.last}"></td>
      <td th:text="|是否偶数次迭代：| + ${iterState.even}"></td>
      <td th:text="|是否奇数次迭代：| + ${iterState.odd}"></td>
   </tr>
</table>
```

代码清单 7-10 使用了#numbers 对象的 sequence 方法，该方法会返回数组。在遍历时设置了两个迭代对象：data 与 iterState，其中 data 为数组节点，iterState 为表示迭代状态的对象，它保存了以上各个变量。为了更好地掌握迭代的方法，我们来看一个常见的需求：在列表中，奇数行的背景色为白色，偶数行的背景色为灰色，则模板实现如下：

```
<table>
   <tr th:each="data,iterState : ${#numbers.sequence(0, 1)}"
      th:style="|background-color: | + ${iterState.odd ? 'white' : 'gray'} + |;|">
      <td th:text="${data}"></td>
   </tr>
</table>
```

以上的代码片断使用了 th:style 属性，也使用了 odd 变量，还使用了条件运算符（三目运算符）。在表达式中会判断 odd 是否为 true，成立则返回“white”，否则返回“gray”。执行以上的代码片断，输出结果如下：

```
<table>
   <tr style="background-color: white;">
      <td>0</td>
   </tr>
   <tr style="background-color: gray;">
      <td>1</td>
   </tr>
</table>
```

除了使用 odd 变量外，还可以换一种写法，使用 index 或者 count 变量来求余，再根据余数来决定返回的颜色，如${(iterState.index % 2 == 0) ? 'white' : 'gray'}。一般情况下，使用 odd 与 event 变量，就可以解决问题，没必要再进行求余。

另外，在前面的例子中，我们显式声明了迭代状态对象：th:each="data, interState : ${array},

仅供非商业用途或交流学习使用

该表达式中的 interState 就是我们显式声明的迭代状态对象，如果不想这样显式声明，也可以使用以下方式获取：

```
<table>
    <tr th:each="data : ${#numbers.sequence(0, 1)}"
        th:style="|background-color: | + ${dataStat.odd ? 'white' : 'gray'} + |;|">
        <td th:text="${data}"></td>
    </tr>
</table>
```

在以上的代码片断中，迭代节点的变量名为“data”，如果要获取迭代状态对象，则使用“dataStat”，获取的规则为“节点变量名 + Stat”。

7.4.6 数据延迟加载

Thymeleaf 支持数据延迟加载，在每一次处理模板时，都可以不必加载数据，是否要加载数据，由模板的逻辑决定。例如，编写 Java 类，将数据设置到 Context 中，请见代码清单 7-11。

代码清单 7-11：codes\07\7.4\iter-test\src\main\java\org\crazyit\boot\c7\LazyTest.java

```
public class LazyTest {

    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        Context ctx = new Context();
        // 设置延迟加载
        ctx.setVariable("users", new LazyContextVariable() {
            @Override
            protected Object loadValue() {
                return getUsers();
            }
        });
        ctx.setVariable("show", false);
        String result = templateEngine.process("lazy.html", ctx);
        System.out.println(result);
    }

    static List<User> getUsers() {
        System.out.println("执行数据查询");
        List<User> users = new ArrayList<User>();
        users.add(new User(1, "Angus"));
        users.add(new User(2, "Paris"));
        return users;
    }
}
```

在该代码清单的粗体字代码中，使用了 LazyContextVariable 对象。这是一个抽象类，需要我们实现 loadValue 方法，在该方法中调用数据查询逻辑，本例是直接调用 getUsers 方法。在实际应用中，可以直接查询数据库。另外还设置了一个 show 变量，该变量用于决定是否查询数据。对应的模板内容，请见代码清单 7-12。

代码清单 7-12：codes\07\7.4\iter-test\src\main\resources\lazy.html

```
<table th:if="${show == true}">
    <tr th:each="user : ${users}">
        <td th:text="${user.name}"></td>
```


仅供非商业用途或交流学习使用

```
    </tr>
</table>
```

在该模板中，如果 show 变量为 true，才会执行 getUsers 方法，如果不使用延迟加载功能，按照之前的写法，不管界面是否显示数据，都会先将 User 集合查询出来。这相对于原来的方法，可以提升性能。

7.4.7 在 Spring Boot 中使用延迟加载

前面是在一般环境中使用延迟加载，在 Spring Boot 环境下同样可以使用该功能。在控制器中设置数据时，使用 LazyContextVariable 即可，控制器代码请见代码清单 7-13。

代码清单 7-13：codes\07\7.4\iter-boot\src\main\resources\templates\lazy.html

```
@SpringBootApplication
@Controller
public class ThyApp {

    public static void main(String[] args) {
        SpringApplication.run(ThyApp.class, args);
    }

    @RequestMapping("/lazy/{show}")
    public String lazy(Model model, @PathVariable("show") boolean show) {
        model.addAttribute("users", new LazyContextVariable() {
            @Override
            protected Object loadValue() {
                return queryUsers();
            }
        });
        model.addAttribute("show", show);
        return "lazy";
    }

    private List<User> queryUsers() {
        System.out.println("执行查询数据");
        List<User> users = new ArrayList<User>();
        users.add(new User(1, "userA"));
        users.add(new User(2, "userB"));
        return users;
    }
}
```

不同的是，这里使用 Model 来设置变量，通过路径变量“show”来决定是否显示数据。该模板内容与前一小节的模板（代码清单 7-12）内容一致。运行代码清单 7-13 启动服务器，在浏览器中访问：http://localhost:8080/lazy/true，由控制台与浏览器的输出可知，queryUsers 方法被执行，而将路径参数改为 false，则不会执行该方法。

7.4.8 星号表达式

前面的大部分模板，都是使用变量表达式（${...}）来显示变量，而除了这种表达式外，还可以使用选定对象表达式（*{...}），也称为星号表达式。两种表达式的使用方式基本一致，但如果在模板中先选定了对象，则需要使用星号表达式。从功能上来讲，星号表达式的功能更多。星号表达式的使用请见代码清单 7-14。

仅供非商业用途或交流学习使用

代码清单 7-14：codes\07\7.4\iter-test\src\main\resources\asterisk.html

```
// 使用变量表达式
<table>
    <tr>
        <td th:text="${user.id}"></td>
        <td th:text="${user.name}"></td>
    </tr>
</table>
// 选定对象后，使用星号表达式
<table>
    <tr th:object="${user}">
        <td th:text="*{id}"></td>
        <td th:text="*{name}"></td>
    </tr>
</table>
```

使用 th:object 属性声明选定“user”变量，然后在“tr”节点的范围内，就可以使用星号表达式。在这里直接使用 User 的属性即可。

在前一章我们学习了 Thymeleaf 的内置对象，其中有一个#object 对象，使用该对象的效果等同于使用星号表达式，下面的代码片断与前面的星号表达式的效果一致：

```
<table>
    <tr th:object="${user}">
        <td th:text="${#object.id}"></td>
        <td th:text="${#object.name}"></td>
    </tr>
</table>
```

我们还可以在循环体中使用星号表达式，如以下的代码片断：

```
<table>
    <tr th:each="u : ${users}" th:object="${u}">
        <td th:text="*{id}"></td>
        <td th:text="*{name}"></td>
    </tr>
</table>
```

在以上的代码片断中，对 users 变量进行循环，且将循环节点设置到名称为“u”的变量中，再使用 th:object 属性选定 User 对象，那么在 tr 下面的 td 节点，就可以直接使用星号表达式来输出了。

再次提醒，在没有选定对象的情况下，变量表达式与星号表达式的作用一致。

7.5 本章小结

本章的内容较为简单，主要讲解了 Thymeleaf 的一些模板语法。如果之前接触过 Web 开发，或者使用过 OGNL、JSTL 等表达式，学习本章的内容会感觉非常轻松。读者在学习过程中，特别要注意一些表达式的使用场合，例如变量表达式（${...}）、文本表达式（#{...}）、星号表达式（*{...}），要知道在什么地方使用什么表达式。

仅供非商业用途或交流学习使用

CHAPTER

8

第8章
深入 Thymeleaf 模板

本章要点

- 模板片断
- 模板逻辑分离
- 模板注释
- 内联语法
- 模板缓存

仅供非商业用途或交流学习使用

前一章讲解了 Thymeleaf 的常用语法，例如运算符、逻辑判断、迭代等，掌握了这些语法后，就可以在模板中处理各种数据了。本章将深入讲解 Thymeleaf 模板，主要讲解模板片断、模板逻辑分离、注释等内容，学习完这些知识后，你能够灵活运用 Thymeleaf 的模板功能。由始至终，Thymeleaf 都在实现一个目标：不破坏原型页面结构，让界面设计人员与程序员都能无缝地参加到研发工作中。在本章，我们同样可以感受到，模板中的很多功能都是为了实现这个目标而存在的。

8.1 模板片断

一个系统的页面中，有很多公共的内容需要提取出来单独存放，例如系统菜单、版权信息等。在使用 JSP 时，可以将这些内容放到一个独立的 JSP 页面中，然后在其他的页面直接使用“include”语法将其包含进来。Thymeleaf 也支持类似的功能，这些独立出来的模板，被称为“模板片断”。

8.1.1 定义与引用模板片断

假如版权信息存放在一个独立的 HTML 模板文件中，如：

```
<span>© 2006-2018 疯狂 Java 联盟版权所有</span>
```

在主页面中，可以使用 th:insert 等属性来引用定义的模板片断，请见代码清单 8-1。

代码清单 8-1：codes\08\8.1\tpl-fragment\src\main\resources\define\index.html

```
<html>
<body>
    <div th:insert="~{define/footer.html}"></div>
</body>
</html>
```

在模板中可以使用片断表达式（~{...}）来定义插入的模板片断。对于以上模板，我们可以理解为，在 index.html 中插入 footer.html 的内容，模板对应的 Java 类请见代码清单 8-2。

代码清单 8-2：codes\08\8.1\tpl-fragment\src\main\java\org\crazyit\boot\c8\DefineTest.java

```
public class DefineTest {

    public static void main(String[] args) {
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        Context ctx = new Context();
        String result = templateEngine.process("define/index.html", ctx);
        System.out.println(result);
    }
}
```

运行代码清单 8-2，输出结果如下：

```
<html>
<body>
    <div><span>© 2006-2018 疯狂 Java 联盟版权所有</span></div>
</body>
</html>
```


仅供非商业用途或交流学习使用

在前面的例子中，仅仅通过 th:insert 属性将整个片断页面（footer.html）的内容插入到主页面（index.html）中。实际上我们可以在公共页面中定义多个片断，再由主页面决定插入哪个。可以使用 th:fragment 属性来定义片断，请见代码清单 8-3。

代码清单 8-3：codes\08\8.1\tpl-fragment\src\main\resources\fragment\footer.html

```
<span th:fragment="copy2017">© 2006-2017 疯狂 Java 联盟版权所有</span>
<span th:fragment="copy2018">© 2006-2018 疯狂 Java 联盟版权所有--狗年快乐</span>
```

codes\08\8.1\tpl-fragment\src\main\resources\fragment\index.html

```
<body>
    <div th:insert="~{fragment/footer.html::copy2018}"></div>
</body>
```

在片断页面（footer.html）中，使用 th:fragment 属性定义了两个模板片断。在主页面（index.html）中使用“~{模板名称::选择器}”语法来插入模板片断，指定插入 footer.html 的“copy2018”片断。执行 index.html 模板后，输出如下：

```
<body>
    <div><span>© 2006-2018 疯狂 Java 联盟版权所有--狗年快乐</span></div>
</body>
```

下面，介绍片断引用的语法。

8.1.2 片断引用语法

前面使用 th:insert 属性插入片断，除此之外，还可以使用 th:replace 和 th:include 属性插入。不管使用哪个属性，都需要提供片断表达式来选择片断。在前一小节的例子中，使用了“~{模板名称}”与“~{模板名称::选择器}”语法来选取片断。除此之外，还可以使用“~{::选择器}”或者“~{this::选择器}”语法来选取模板中的节点，请见代码清单 8-4。

代码清单 8-4：codes\08\8.1\tpl-fragment\src\main\resources\this\index.html

```
<body>
    <div th:insert="~{::copy}"></div>
    <div th:insert="~{this::copy}"></div>

    <span th:fragment="copy">index 页面的版权信息</span>
</body>
```

在代码清单 8-4 中使用了两次 th:insert 属性，一次使用“~{::选择器}”表达式，一次使用“~{this::选择器}”表达式，二者的作用一致，都是找到本模板中的“copy”片断并进行插入。

在片断表达式中使用 this 语法时，有一点需要注意，如果无法在当前模板中找到该片断，则最终会到 TemplateEngine 的 process 方法处理的模板中寻找。模板与片断页面请见代码清单 8-5。

代码清单 8-5：codes\08\8.1\tpl-fragment\src\main\resources\process\index.html

```
<body>
    <div th:insert="~{process/footer.html}"></div>
    <span th:fragment="copy">index 页面的版权信息</span>
</body>
```

codes\08\8.1\tpl-fragment\src\main\resources\process\index.html

```
<span th:insert="~{this::copy}"></span>
```


仅供非商业用途或交流学习使用

TemplateEngine 的 process 方法最终会处理 index.html 模板，该模板使用 th:insert 属性插入 footer.html。在 footer.html 中会使用“~{this:copy}”表达式来插入本模板的“copy”片断，但是 footer.html 中并不存在“copy”片断，所以最后到 index.html 中寻找“copy”片断。处理 index.html 模板后，结果输出如下：

```
<body>
    <div><span><span>index 页面的版权信息</span></span></div>
    <span>index 页面的版权信息</span>
</body>
```

上面的例子绕了一个圈，想要告诉读者的是，当使用“~{this::选择器}”语法，无法在当前模板中找到片断时，会到 process 方法处理的模板中寻找片断。本例子中 process 方法处理的是 index.html 模板，而在 footer.html 模板中使用了 this 语法，这样无法找到 copy 片断，最终会使用 index.html 模板中的 copy 片断。最后总结一下片断引用的语法：

- ~{模板名称}
- ~{模板名称::选择器}
- ~{this::选择器}或~{::选择器}

8.1.3 选择器的基础语法

模板选择器允许我们在引用的模板中，选取需要的 HTML 片断。片断选择器的语法类似于 XPath、CSS 与 JQuery 选择器的语法。片断选择器的基本语法如下。

- 第 1 种：/name，选中直接子节点中节点名称为 name 的节点，如果无法选中则直接抛出异常。
- 第 2 种：//name，选中全部子节点中节点名称为 name 的节点，选择时不考虑节点位置。
- 第 3 种：name[@attribute=’value’]，在全部节点中，选取名称为 name 并且属性值等于“value”的节点。如果要根据多个属性值选取节点，则可以使用 and 连接，例如，name[@attr1=’value1’ and @attr2 = ‘value2’]，还可以省略 and，以多个中括号代替，例如，name[@attr1=’value1’][@attr2 = ‘value2’]。
- 第 4 种：//name[@attr=’value’][index]，如果根据节点名称或者属性选取出来的结果中包含多个节点，则可以在后面加上“[index]”来指定节点索引。

以上几种语法的使用请见代码清单 8-6。

代码清单 8-6：codes\08\8.1\tpl-fragment\src\main\resources\select\footer.html

```
/************第 1 种语法***********/
<div id="footerA">footer A</div>
/************第 2 种语法***********/
<div>
    <div id="footerB"></div>
</div>
/************第 3 种语法***********/
<div>
    <span class="content">footer c</span>
    <span class="content">footer d</span>
</div>
```


仅供非商业用途或交流学习使用

```
        <div>
            <span class="andContent" id="testAnd1">test and 1</span>
            <span class="andContent" id="testAnd2">test and 2</span>
        </div>
        /************第 4 种语法***********/
        <span class="footerG">footer g1</span>
        <span class="footerG">footer g2</span>
```

codes\08\8.1\tpl-fragment\src\main\resources\select\index.html

```
        /************第 1 种语法***********/
        //选中直接子节点 id 为 footerA 的 div
        <div th:insert="~{select/footer.html :: /div[@id='footerA']}"></div>
        /************第 2 种语法***********/
        // 选中全部子节点中 id 为 footerB 的 div
        <div th:insert="~{select/footer.html :: //div[@id='footerB']}"></div>
        /************第 3 种语法***********/
        // 选中 class 为 content 的 span 节点
        <div th:insert="~{select/footer.html :: span[@class='content']}"></div>
        // 选中 class 为 andContent，并且 id 等于 testAnd1 的节点
        <div th:insert="~{select/footer.html :: span[@class='andContent' and @id=
'testAnd1']}"></div>
        /************第 4 种语法***********/
        // 选中 class 为 footerG 的 span（有多个），选出第一个
        <div th:insert="~{select/footer.html :: //span[@class='footerG'][0]}"></div>
```

代码清单 8-6 中的 index.html 使用几种选择器语法，引用 footer.html 模板中的片断。以上的语法使用较为简单，我们可以将它们应用在稍微复杂一点的场景中，例如有以下的片断：

```
        <div class="footerContent">
            <div>
                <span id="footerE">footer e</span>
                <span id="footerF">footer f</span>
            </div>
        </div>
```

我们要选择 id 为“footerE”的 span，那么选择器可以有以下的写法：

```
        <div  th:insert="~{select/footer.html  ::  //div[@class='footerContent']/div/
span[@id='footerF']}"></div>
        <div   th:insert="~{select/footer.html   ::   //div[@class='footerContent']//
span[@id='footerF']}"></div>
```

以上的代码片断使用了两种语法，第一行使用了第 1 种语法（/name），第二行使用了第 2 种语法（//name）。这两种写法的效果一致，都会先选中 class 属性为“footerContent”的 div 节点，再选中该 div 下 id 为“footerF”的 span 节点。

除此之外，还有两个小细节需要注意：

第一，第 2 种语法（//name）与第 3 种语法（name）实际上是等价的，除了可以选择 HTML 的节点外，使用这两种语法还可以选中使用了 th:ref、th:fragment 属性的节点，例如以下的片断：

```
        <div th:ref="footerRef">
            <span>footer ref</span>
        </div>
```

以下的两种写法，都可以选中上面的 span 节点：

仅供非商业用途或交流学习使用

```
    <div th:insert="~{select/footer.html :: //footerRef/span}"></div>
    <div th:insert="~{select/footer.html :: footerRef/span}"></div>
```

虽然这两种写法的效果一样，使用起来都比较方便，但是笔者建议，为了增强代码的可读性，对于 HTML 节点，尽量使用“//name”语法，而对于使用了 th:ref、th:fragment 属性的节点，还是使用第 3 种语法（name）较好。

第二，在选取时可以不提供节点名称，直接将“//div[@class='footer']”写为“[@class='footer']”，后一种写法，选择器会查找所有 class 为“footer”的节点。

8.1.4 选择器的其他语法

选择器还存在其他的语法，实际上这些语法是基础语法的简化，主要包括以下的写法。

- **省略“@”符号：**在选择属性时，需要使用@符号，例如“div[@id='footerA']”，可以简写为“div[id='footerA']”。当根据多个属性选择时，也可以省略@符号。
- **使用“#”符号代替 id 选取：**“div#id”等价于“div[id='id']”。
- **使用“.”符号代替类选取：**“div.class”等价于“div[class='class']”，也可以直接写成“.class”，即省略节点名称。
- **使用“%”代替片断引用：**如果片断中的节点使用了 th:ref 或 th:fragment 属性，则可以使用“div%ref”语法来选取节点，当然也可以省略节点名称。
- **混合使用：**在同一句选择器的语句中，既可以使用最基础的语法，也可以使用省略语法。

以上的几种写法，请见代码清单 8-7。

代码清单 8-7：codes\08\8.1\tpl-fragment\src\main\resources\os\index.html

```
    /************省略@***********/
    <div th:insert="~{os/footer.html :: /div[id='footerA']}"></div>
    <div th:insert="~{os/footer.html :: span[class='andContent'][id='testAnd1']}">
</div>
    /************使用#代替 id 选取***********/
    <div th:insert="~{os/footer.html :: div#footerA}"></div>
    /************使用.代替 class 选取***********/
    <div th:insert="~{os/footer.html :: span.footerClass}"></div>
    <div th:insert="~{os/footer.html :: .footerClass}"></div>
    /************使用%代替 th:ref 节点的引用***********/
    <div th:insert="~{os/footer.html :: div%footerRef}"></div>
    <div th:insert="~{os/footer.html :: %footerRef}"></div>
    /************混合使用***********/
    <div th:insert="~{os/footer.html :: div.footerContent//span[id='footerE']}">
</div>
```

由前面的例子可知，Thymeleaf 的片断选择器可以极大地方便我们选取片断节点。实际上，不管是使用基础的写法，还是使用省略写法，效果都一致。如果你做的是个人项目，则可以自由发挥，自己看懂就好了，但如果是团队开发项目，特别是一些大型项目，笔者建议，应当统一代码风格。

8.1.5 insert、replace 与 include 的区别

在前面的例子中，都是使用 th:insert 属性来插入模板片断，除了 insert 外，也可以使用

仅供非商业用途或交流学习使用

replace 与 include 来引用模板片断。下面通过例子来对比这三个属性，模板与片断页面请见代码清单 8-8。

代码清单 8-8：codes\08\8.1\tpl-fragment\src\main\resources\compare\footer.html

```
    <span th:fragment="copy">footer text</span>
```

codes\08\8.1\tpl-fragment\src\main\resources\compare\index.html

```
<body>
    <div th:insert="~{compare/footer.html::copy}"></div>
    <div th:replace="~{compare/footer.html::copy}"></div>
    <div th:include="~{compare/footer.html::copy}"></div>
</body>
```

在片断中定义了一个普通的 span 节点，在模板中分别使用 insert、replace 与 include 来使用该片断。执行 index.html 后，输出结果如下：

```
<body>
    <div><span>footer text</span></div>
    <span>footer text</span>
    <div>footer text</div>
</body>
```

根据结果可知，th:insert 属性会将选择到的 span 节点插入 div 中，th:replace 属性会将原来的 div 节点替换为 span 节点，最后的 th:include 属性，则会将 span 节点的内容（包括子节点）插入 div 中。其中 insert 与 include 有点类似，但仔细看就知道它们的区别，insert 包括本节点，include 则只插入节点下面的内容及子节点，并不包括本节点。

8.1.6 含有变量的片断引用

有时需要在片断中使用一些传入的变量，这些变量将会由主模板传入，引用片断的语法为“~{模板::fragment 名称(参数)}”。例如，代码清单 8-9 定义了一个片断及一个模板。

代码清单 8-9：codes\08\8.1\tpl-fragment\src\main\resources\param\footer.html

```
    <div th:fragment="welcome(user, company)" >
        <span th:text="|Hello | + ${user} + |, welcome to | + ${company}"></span>
    </div>
```

codes\08\8.1\tpl-fragment\src\main\resources\param\index.html

```
<body>
    /************使用常量传参***********/
    <div th:replace="~{param/footer.html::welcome('Angus', 'Crazyit')}"></div>
    /************使用变量传参***********/
    <div                    th:replace="~{param/footer.html::welcome(${userParam},
${companyParam})}"></div>
</body>
```

在 footer.html 的片断中定义了两个参数：user 与 company，该片断只会简单地将变量放到 span 中。在模板 index.html 中使用了两次 th:replace 来引用 footer.html 的片断，第一次传入普通的常量，第二次传入变量，它们都会被设置到 Context 中，运行代码请见代码清单 8-10。

代码清单 8-10：codes\08\8.1\tpl-fragment\src\main\java\org\crazyit\boot\c8\ParamTest.java

```
public class ParamTest {

    public static void main(String[] args) {
```

```
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        Context ctx = new Context();
        ctx.setVariable("userParam", "Paris");
        ctx.setVariable("companyParam", "CE");
        String result = templateEngine.process("param/index.html", ctx);
        System.out.println(result);
    }
}
```

运行代码清单 8-10，输出结果如下：

```
<body>
    /************使用常量传参***********/
    <div >
        <span>Hello Angus, welcome to Crazyit</span>
    </div>
    /************使用变量传参***********/
    <div >
        <span>Hello Paris, welcome to CE</span>
    </div>
</body>
```

由于在片断中定义了两个参数，因此如果在使用 th:replace 等属性时传入不符的参数，那么在处理模板时就会抛出异常。不传入参数也会抛出异常，请见以下语句：

```
<div th:replace="~{param/footer.html::welcome()}"></div>
```

想避免这种情况，可以在定义片断时，不显式指定片断参数，例如以下的代码片断：

```
    <div th:fragment="welcomeTwo" >
        <span th:text="|Hello | + ${val1} + |, welcome to | + ${val2}"></span>
    </div>
```

那么在引用片断时，不管是否传入参数，都不会抛出异常，例如以下代码片断：

```
<div th:replace="~{param/footer.html::welcomeTwo(val1='ab', val2='cd')}"></div>
<div th:replace="~{param/footer.html::welcomeTwo}"></div>
<div th:include="~{param/footer.html::welcomeTwo}"
    th:with="val1=${userParam},val2=${companyParam}"></div>
```

以上分别引入了三次片断，第一次引入时直接传入相应的参数，第二次引入并没有传入任何参数，第三次使用了 th:include 来引入片断。由于在定义片断时，并没有显式指定片断参数，因此不管是否传入参数，都不会抛出异常。注意，在第三次引入片断时，使用 th:with 属性来定义传入的参数，这两个参数会作为本地变量存在。如果使用 replace，则在片断中将无法获取 val1 与 val2 这两个变量，使用 include 与 insert 则可以获取到，这是由于 repalce 会替换掉原来的 div，而使用 include 与 insert 时，原来的 div 仍然存在，只是插入的内容不同而已。

注意：定义片断时是否要设计参数，由具体情况而定。大家需要知道，在引入含有显式参数的片断时，如果不传入相应的参数，会抛出异常，如果不显式指定参数则不会抛出异常。

➤➤ 8.1.7　片断块引用

我们可以使用 th:block 定义代码块，例如在一个循环体内，如果想对 tr（表格行）进行循环，则可以直接在 tr 里面使用 th:each 属性，如果不想把该属性写到 tr 中，则可以添加代码块，

如以下代码：

```
<table>
    <th:block th:each="number : ${#numbers.sequence(0, 3)}">
    <tr>
        <td th:text="${number}"></td>
    </tr>
    </th:block>
</table>
```

以上代码的效果，与在 tr 中添加 th:each 属性一致。th:block 是一个属性容器，可以在其中添加任何的 th 属性，而对于 th:block 本身来说，它并无特殊的含义。如果在片断中定义 th:block，则可以形成片断块，请见代码清单 8-11。

代码清单 8-11：codes\08\8.1\tpl-fragment\src\main\resources\block\header.html

```
<head th:fragment="common_header(links)">
    <link rel="stylesheet" type="text/css" media="all" href="/common.css">
    <th:block th:replace="${links}" />
</head>
```

代码清单 8-11 是一个 HTML 头，里面定义了一个公共的 CSS。在引用该片断的页面中，只需要传入自己模板的 CSS 模板，就可以形成一个完整的 CSS 片断。例如在销售模块页面中，使用 common.css+sale.css，则在引用以上模板时，只需要传入 sale.css 的 link 即可。上面片断中的 th:block 会被替换为具体的 CSS 引用，请见代码清单 8-12。

代码清单 8-12：codes\08\8.1\tpl-fragment\src\main\resources\block\header.html

```
<body>
    <head th:replace="block/header.html :: common_header(~{this :: link})">
        <link rel="stylesheet" href="/css/sale.css">
    </head>
</body>
```

以上模板自己本身就有一个 sale.css，然后使用 th:replace 来引用前面的片断，拿片断中的 head 替换其本身的 head，且将自己的 sale.css 引用传入片断，最后的处理结果如下：

```
<body>
    <head>
        <link rel="stylesheet" type="text/css" media="all" href="/common.css">
        <link rel="stylesheet" href="/css/sale.css">
    </head>
</body>
```

实际上，在片断文件中定义了一个代码块，该代码块的内容完全由引用的模板决定，最终生成的内容由引用片断的模板决定。

8.1.8 无操作符的使用

使用前面的片断块方式可以实现 HTML 页面标题的动态更换，请见代码清单 8-13。

代码清单 8-13：codes\08\8.1\tpl-fragment\src\main\resources\block\header.html

```
<head th:fragment="noop(title)">
    <title th:replace="${title}">片断的标题</title>
    <link rel="stylesheet" type="text/css" media="all" href="/common.css">
</head>
```

以上模板定义了一个含有标题的 head，其会根据传入的 title 值来生成 title 节点。模板在引用时，传入自己的 title 即可，请见代码清单 8-14。

代码清单 8-14：codes\08\8.1\tpl-fragment\src\main\resources\block\index.html

```
<head th:replace="block/header.html :: noop(~{this :: title})">
    <title>模板的标题</title>
</head>
```

处理 index.html 模板后，生成结果如下：

```
<head>
    <title>模板的标题</title>
    <link rel="stylesheet" type="text/css" media="all" href="/common.css">
</head>
```

在操作时，如果不想将本页面的模板替换到片断中，则可以传入无操作符：“_”，如果传入无操作符，最后会使用片断的 title。我们也可以传入空的片断：“~{}”，这样最后不会生成 title 节点，如以下的代码：

```
<head th:replace="block/header.html :: noop(~{})">
    <title>模板的标题</title>
</head>
<head th:replace="block/header.html :: noop(_)">
    <title>模板的标题</title>
</head>
```

以上的两个模板，第一个传入了空的片断，最后将不会生成 title 节点；第二个传入了无操作符，则会使用片断中的 title 节点生成结果。处理以上的模板，生成的结果如下：

```
<head>

    <link rel="stylesheet" type="text/css" media="all" href="/common.css">
</head>
<head>
    <title>片断的标题</title>
    <link rel="stylesheet" type="text/css" media="all" href="/common.css">
</head>
```

上面通过一个更换 title 节点的例子来展示如何使用空片断与无操作符，读者在使用时注意这两者的区别。

8.1.9 删除模板

在实际开发中，我们经常会遇到一些特殊的列表页面，例如在一个产品展示的列表界面，前面的两个产品需要加上“推荐”字样，模板请见代码清单 8-15。

代码清单 8-15：codes\08\8.1\tpl-fragment\src\main\resources\remove\index.html

```
<ul>
    <li th:each="p : ${products}">
        <div th:text="${p.name}">商品名称</div>
        <div th:if="${pStat.index <= 1}"><span style="color: red;">推荐</span></div>
    </li>
</ul>
```

其中的粗体字代码表示，当索引小于等于 1 时，会显示“推荐”的 div。对于这种模板，我们也可以使用 th:remove 来实现，当索引大于 1 时，删除“推荐”的 div。以下模板的效果与代码清单 8-15 一致：

```
<ul>
    <li th:each="p : ${products}">
        <div th:text="${p.name}">商品名称</div>
        <div th:remove="${pStat.index > 1} ? all : none"><span style="color: red;">
推荐</span></div>
    </li>
</ul>
```

这里使用了 th:remove 属性，当索引大于 1 时，th:remove 取值为“all”，否则取值为“none”。该段表达式的意思为，在迭代集合时，若索引大于 1，则将整个 div（包括其子节点）删除，否则不进行任何操作，也就是会在产品列表的前面两个商品上，加上“推荐”字样。th:remove 属性支持以下值。

- all：在处理模板时把当前 th:remove 所在的节点删除，包括其子节点。
- body：将当前节点的全部子节点删除，例如前面的例子，会将 div 下面的 span 删除，而本节点则不会被删除。
- tag：将所在的节点删除，不删除子节点，例如前面的例子，会删除 div 节点，而保留 span 节点。
- all-but-first：删除当前节点下面的全部子节点，但会保留第一个。
- none：不进行任何操作，表达式中常使用该值。

将数据设置到 Context 中，请见代码清单 8-16。

代码清单 8-16：codes\08\8.1\tpl-fragment\src\main\java\org\crazyit\boot\c8\RemoveTest.java

```
TemplateEngine templateEngine = new TemplateEngine();
templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
Context ctx = new Context();
List<Product> ps = new ArrayList<Product>();
ps.add(new Product("疯狂 Java 讲义"));
ps.add(new Product("疯狂 Spring Cloud"));
ps.add(new Product("疯狂工作流讲义"));
ctx.setVariable("products", ps);
String result = templateEngine.process("remove/index.html", ctx);
System.out.println(result);
```

运行代码清单 8-16，控制台输出如下：

```
<ul>
    <li>
        <div>疯狂 Java 讲义</div>
        <div><span style="color: red;">推荐</span></div>
    </li>
    <li>
        <div>疯狂 Spring Cloud</div>
        <div><span style="color: red;">推荐</span></div>
    </li>
    <li>
```

```
            <div>疯狂工作流讲义</div>
        </li>
    </ul>
```

根据结果可知，前面两条数据都会被加上“推荐”的 div，而从第 3 条数据开始，则会“删除”这个 div。

8.1.10 基本表达式总结

至此，我们已经学习完了全部 Thymeleaf 的基本表达式，包括文本表达式（#{...}）、变量表达式（${...}）、链接表达式（@{...}）、星号表达式（*{...}）与片断表达式（~{...}）。在此，简单总结一下这几个表达式的使用方法，请见代码清单 8-17。

代码清单 8-17：codes\08\8.1\tpl-fragment\src\main\resources\exp\index.html

```
/************ 变量表达式 ************/
<div th:text="${product.name}"></div>
/************ 文本表达式 ************/
<div th:text="#{myKey}"></div>
<div th:text="${#messages.msg('myKey')}"></div>
/************ 链接表达式 ************/
<a href="view.html" th:href="@{~/order/details}"></a>
/************ 星号表达式 ************/
<li th:object="${product}">
    <span th:text="*{name}"></span>
    <span th:text="${#object.name}"></span>
</li>
/************ 片断表达式 ************/
<div th:insert="~{exp/footer.html}"></div>
```

这些表达式的使用方法，在相应的章节中都做了具体的讲述，例如文本表达式在 6.1.1 节讲述，链接表达式在 6.1.7 节讲述，星号表达式在 7.4.8 节讲述。对于这些表达式，我们要做到，看到相应的表达式，知道其含义，至于更详细的解释，可查阅相应的章节。

8.2 模板与逻辑分离

在前面的 Thymeleaf 模板中，都是将 Thymeleaf 的属性写到 HTML 节点中，这些带有逻辑的属性与 HTML 节点混合在一起。Thymeleaf 提供了一种更为彻底的模板与逻辑分离的方案，且对 HTML 文件不会有任何的破坏与影响，这一节我们就来学习该方案。

8.2.1 逻辑分离例子

实现模板与逻辑彻底分离，不用对原有的模板文件作任何修改，在此基础上提供另外一个 XML 文件，用来保存程序逻辑。下面使用一个简单的例子来说明该功能，模板内容请见代码清单 8-18。

代码清单 8-18：codes\08\8.2\decouple\src\main\resources\decouple\index.html

```
<table id="dataTable" class="content">
    <tr>
        <td>1</td>
        <td>Angus</td>
    </tr>
</table>
```

假如代码清单 8-18 是页面同事提供的原型文件，按照前面的方式，我们一般会直接在该文件中写入 Thymeleaf 的属性来实现分离。而现在我们可以将逻辑写到另外一个 XML 文档中，请见代码清单 8-19。

代码清单 8-19：codes\08\8.2\decouple\src\main\resources\decouple\index.th.xml

```
<?xml version="1.0"?>
<thlogic>
    <attr sel="[id='dataTable']">
        <attr sel="tr" th:each="user : ${users}">
            <attr sel="td[0]" th:text="${user.id}"></attr>
            <attr sel="td[1]" th:text="${user.name}"></attr>
        </attr>
    </attr>
</thlogic>
```

代码清单 8-19 是一个普通的 XML 文档，实现逻辑分离的 XML 并没有相关规范，可以将根节点 thlogic 看作一个逻辑块，它下面可以包含一个或者多个 arrt 节点，attr 节点下也可以出现 0 个或者多个 attr 节点，由于缺少规范约束，因此节点部署比较随意，只要符合 XML 文档最基本的规范（例如要关闭节点），就不会出现异常。唯一要注意的是，每个 attr 节点，都需要提供 sel 属性，该属性用于表示 attr 节点选中（或对应）哪一个模板节点，如果不提供 sel 属性，则在处理模板时会抛出异常信息：“org.thymeleaf.exceptions.Template- InputException: Error while processing decoupled logic file: <attr> injection tag does not contain any "sel" selector attributes.”。

代码清单 8-19 的粗体字代码表示，该 attr 选中了 id 为“dataTable”的 HTML 节点，也可写成“#dataTable”，也可以根据 class 来选取“.className”。sel 属性的语法与在前面章节中介绍的选择器语法类似。

接下来，编写运行代码，处理模板文件，请见代码清单 8-20。

代码清单 8-20：codes\08\8.2\decouple\src\main\java\org\crazyit\boot\c8\DecoupleTest.java

```
public class DecoupleTest {

    public static void main(String[] args) {
        // 设置模板解析器
        ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
        // 设置使用逻辑分离
        resolver.setUseDecoupledLogic(true);
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(resolver);
        // 设置集合参数
        Context ctx = new Context();
        List<User> users = new ArrayList<User>();
        users.add(new User(1, "Angus"));
        users.add(new User(2, "Paris"));
        ctx.setVariable("users", users);
        String result = templateEngine.process("decouple/index.html", ctx);
        System.out.println(result);
    }
}
```

这个运行代码与前面章节中的运行代码有点区别，需要设置模板解析器开启逻辑分离功能，其他并无太大区别。实现逻辑分离实际上并不复杂，仅仅是将程序逻辑放到一个特定的 XML

文档中，然后在 XML 中使用 attr 节点，将逻辑与模板中的 HTML 节点进行关联。

8.2.2 逻辑分离配置

如前面的逻辑分离例子所示，假如模板存放于类路径下的 decouple 目录中，名称为 test.html，那么在默认情况下，Thymeleaf 会到 decouple 目录下寻找 test.th.xml 文件作为逻辑文件。逻辑文件的存放目录及后缀，可以通过代码进行配置。代码清单 8-21 为运行代码。

代码清单 8-21：codes\08\8.2\decouple\src\main\java\org\crazyit\boot\c8\ConfigTest.java

```
public class ConfigTest {

    public static void main(String[] args) {
        // 设置模板解析器
        ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
        // 设置使用逻辑分离
        resolver.setUseDecoupledLogic(true);
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(resolver);
        StandardDecoupledTemplateLogicResolver logicResolver =
                (StandardDecoupledTemplateLogicResolver) templateEngine
                .getDecoupledTemplateLogicResolver();
        // 设置前缀与后缀
        logicResolver.setPrefix("../logics/");
        logicResolver.setSuffix(".abc.xml");
        // 设置集合参数
        Context ctx = new Context();
        ctx.setVariable("user", "Angus");
        String result = templateEngine.process("config/index.html", ctx);
        System.out.println(result);
    }
}
```

代码清单 8-21 中的粗体字代码表示，获取 TemplateEngine 的逻辑分离处理对象，通过该对象来设置相应的前缀与后缀。

在本例中将前缀设置为“../logics/”，由于此处处理的模板是 config 目录下的 index.html，因此配置为“../logics/”。Thymeleaf 会到 config 目录的上一级目录下，寻找 logics 目录，也就是与 config 同级的 logics 目录，在该目录中存放逻辑文件。本例设置了后缀为“.abc.xml”，因此 Thymeleaf 会到 logics 目录下寻找“index.abc.xml”文件。本例的模板文件与逻辑文件内容，请见代码清单 8-22。

代码清单 8-22：codes\08\8.2\decouple\src\main\resources\config\index.html

```
<span id="span">Hello, my friend. </span>
```

codes\08\8.2\decouple\src\main\resources\logics\index.abc.xml

```
<?xml version="1.0"?>
<thlogic>
    <attr sel="[id='span']" th:text="|Hello, | + ${user}"></attr>
</thlogic>
```

运行代码清单 8-21，控制台输出如下：

```
<span id="span">Hello, Angus</span>
```

根据结果可知，逻辑文件已经被成功读取。在此再次说明，在默认情况下，Thymeleaf 会到模板文件所在目录下寻找“templateName.th.xml”文件，可通过设置逻辑分离对象（IDecoupledTemplateLogicResolver 实现类）来改变这些配置。

8.2.3 关联节点的选择

在逻辑文件中，使用 attr 节点的 sel 属性来关联模板的 HTML 节点，sel 属性的语法与在前面介绍的片断选择器的语法类似，它们都使用 AttoParser 项目进行节点选择。代码清单 8-23 使用不同的语法来进行节点选择。

代码清单 8-23：codes\08\8.2\decouple\src\main\resources\ref\index.th.xml

```
<?xml version="1.0"?>
<thlogic>
    <attr sel="/table/tr/td/span" th:text="${user}"></attr>
    <attr sel="/div/#spanId" th:text="${user}"></attr>
    <attr sel="/div/.spanClass" th:text="${user}"></attr>
    <attr sel="/li/[class = 'spanClass']" th:text="${user}"></attr>
</thlogic>
```

第一个 attr 根据节点的层级选择 span 节点，第二个 attr 选中 div 下 id 为“spanId”的 span 节点，第三与第四个 attr 都是根据 class 来选择 span。这四个 attr 节点在关联了 span 后，会使用 th:text 属性设置文本。除此之外，我们还可以在模板文件中使用 th:fragment 或者 th:ref 属性，然后在逻辑文件中使用这些定义好的属性，请见下面的模板文件内容：

```
<span th:ref="testRef">Prototype content</span>
<span th:ref="testFrag">Prototype content</span>
```

对应的逻辑文件内容如下：

```
    <attr sel="testRef" th:text="${user}"></attr>
    <attr sel="testFrag" th:text="${user}"></attr>
```

在模板中使用 th:ref 或 th:fragment 属性的意义在于，当模板没有 id 或者 class 等属性时，我们可以通过这两个属性，很精确地定位到 HTML 节点。但这样做也有缺点，我们向模板中添加了逻辑（th:ref 与 th:fragment），破坏了原来的模板。

实际上模板与逻辑分离有利也有弊，其优点在于：第一，不会破坏原型界面，这对于设计界面的同事来说是很棒的，因为其工作成果并没有被改动；其次，在功能开发时，分工更加明确，逻辑文件由程序员维护。但这样的分离同样存在弊端，在一定程度上，会增加开发与维护成本。首先，当加入 Thymeleaf 的属性时，就要额外添加 XML 节点，这增加了代码量；其次，在修改某个功能时，程序员需要找到两个文件，并且要对它们进行对比，找到对应的 HTML 节点与逻辑节点才能进行修改，这相对于将逻辑写到模板中，修改起来就没有那么直观了。

笔者的建议是，对于一些界面设计人员参与度比较高的项目，可以考虑使用逻辑分离，如果界面设计人员参与度不高，则将逻辑写到模板中可能更好。

8.3 模板注释

这一节我们讲解模板注释。

8.3.1 解析层注释

一般的 HTML（XML）文档可以使用“<!--内容 -->”进行注释。虽然 Thymeleaf 模板也是一个 HTML 文档，但由于模板会被 Thymeleaf 处理，因此也要对注释进行特别处理。例如，代码清单 8-24 就是 Thymeleaf 模板的解析层注释。

代码清单 8-24：codes\08\8.3\comment\src\main\resources\comment.html

```
<!--/* 这里的内容在解析后将不会显示 */-->
<span th:text="Angus">Prototype Angus</span>
```

所谓解析层注释，就是在被模板引擎解析后，不会显示的注释。代码清单 8-24 中的注释，如果在浏览器中打开，仍然可以显示，但一旦被模板引擎执行，在结果中将看不到这些内容。处理代码清单 8-24 后，结果如下：

```
<span>Angus</span>
```

在 Web 应用中使用解析层注释，最终在浏览器中查看网页源代码，也无法查看到注释的内容。除此之外，它还有一个明显的优点。例如，以下是一个界面设计人员提供的原型 HTML：

```
<table>
    <tr>
        <td>id: 100</td>
        <td>名称：小明</td>
    </tr>
    <tr>
        <td>id: 200</td>
        <td>名称：小红</td>
    </tr>
</table>
```

制作原型的同事一般会添加多条数据来查看效果，而程序员在收到原型页面时，会选取一条数据进行迭代，然后将后面的“模拟数据”删除，例如，加入逻辑后的模板代码如下：

```
<table>
    <tr th:each="user : ${users}">
        <td th:text="${user.id}">id: 100</td>
        <td th:text="${user.name}">名称：小明</td>
    </tr>
</table>
```

这样做就会导致原型文件被修改（删除），为了减少这种对原型页面的破坏，可以使用解析层注释：

```
<table>
    <tr th:each="user : ${users}">
        <td th:text="${user.id}">id: 100</td>
        <td th:text="${user.name}">名称：小明</td>
    </tr>
    <!--/*
    <tr>
        <td>id: 200</td>
        <td>名称：小红</td>
    </tr>
    */-->
</table>
```

在以上的模板代码中，对第二条数据进行注释，这样模板引擎进行解析后，就不会将该条数据展示出来，这很好地保护了原型界面。

8.3.2 原型注释

所谓原型注释，就是指被注释的内容，仍然可以被模板引擎解析并处理，但如果只通过浏览器打开（界面设计人员浏览），则不会显示。使用“<!--/*/ 注释内容 /*/-->”语法进行注释，请见代码清单 8-25。

代码清单 8-25：codes\08\8.3\comment\src\main\resources\comment.html

```
<!--/*/
<span th:text="Angus">Prototype Angus</span>
/*/-->
```

代码清单 8-25 的这段模板代码将会被模板引擎解析，最后会正常输出，但如果使用浏览器打开（双击或拖动到浏览器中打开），则不显示出来，浏览器会认为它已经被注释掉。由此可以看到，Thymeleaf 的设计都是希望能不妨碍到页面设计人员、开发人员的工作，不破坏各个角色的工作成果。

8.4 内联语法

内联是“inlining”的翻译，可以理解为直接在模板中插入文本，是显示文本的另外一种方式。

8.4.1 使用内联语法显示文本

在前面章节中，要在 HTML 中显示文本，一般会使用 th:text 或 th:utext 属性，但其实也可以使用内联语法来显示文本，如以下的代码片断：

```
<div>
   Hello, <span th:text="${user}">Test</span>
</div>
<div>
   Hello, [[${user}]]
</div>
```

以上的代码片断使用了 th:text 属性，也使用了“[[…]]”语法，这两种写法的结果一致。除此之外，还有“[(…)]”语法，其效果与 th:utext 属性一致。这实际上很好理解，将“[[…]]”当作 th:text 来使用，将“[(…)]”当作 th:utext 来使用。

可能有朋友觉得奇怪，原来已经有 th:text 属性了，为什么还要增加一个内联语法呢？因为内联语法比 th:text 简便，那是不是 th:text 已经没有存在的必要了？非也，在 HTML 模板中，我们可以使用两种文本显示方法，但是对于静态的 HTML 文件（界面设计人员使用时），使用内联语法显得不那么方便。如果模板不是 HTML 类型，而是普通的文本、JavaScript 或 CSS 代码，则使用内联语法较为合适，因为在这类文本中，并不存在 XML 节点。换言之，两种文本显示方式，都有各自应用的场景。

可以关闭内联语法，在节点中使用 th:inline=”none”即可：

```
<p th:inline="none"> [[1, 2, 3], [4, 5]]</p>
```

输出结果为：

```
<p>[[1, 2, 3], [4, 5]]</p>
```

这里的 p 节点，如果不为其加入“th:inline”属性，其内容会被当作内联语法的表达式被解析，而且在解析模板时会得到异常：“org.thymeleaf.exceptions.TemplateProcessingException: Could not parse as expression”。

8.4.2 在 JavaScript 中使用内联语法

由前所述，在 JavaScript 中比较适合使用内联语法。在某些情况下，一些数据不会作为 JavaScript 参数，也不会被写死在代码中，而可能会在渲染模板时设置它，此时就可以使用内联语法来处理，请见代码清单 8-26。

代码清单 8-26：codes\08\8.4\inlining\src\main\resources\js.html

```
<head>
    <script th:inline="JavaScript">
        function show() {
            alert('[[${user}]]');
        }
    </script>
</head>
<body>
    <input type="button" value="click me" onClick="show()"/>
</body>
```

代码清单 8-26 中的 JavaScript 的 show 方法，会将变量 user 输出。对于这种情况，可以使用内联语法，并在处理模板时，将 user 变量“放到”JavaScript 语句中。如果在 JavaScript 中不使用内联语法，为了实现上面的效果，可能需要将代码变成以下的样子：

```
<head>
    <script th:inline="JavaScript">
        function show(user) {
            alert(user);
        }
    </script>
</head>
<body>
    <input type="button" value="click me" th:attr="onClick=|show('| + ${user} + |')|" onClick="show()"/>
    // 或者
    <input type="button" value="click me" th:onclick="|show('| + ${user} + |')|" onClick="show()"/>
</body>
```

首先将 JavaScript 中的 show 方法的 alert 内容变为动态的，然后在调用 show 方法的地方，将变量通过表达式传入。这样做不仅增加了代码的复杂度，而且对于原型页面也有较大的改变，特别是对原有的 JavaScript 代码。

8.4.3 在 CSS 中使用内联语法

与 JavaScript 类似，我们也可以在 CSS 中使用内联语法，请见代码清单 8.27。

代码清单 8-27：codes\08\8.4\inlining\src\main\resources\css.html

```
<style>
    .my-text{
        color: [[${color}]];
    }
</style>
<body>
    <span class="my-text">Hello World!</span>
</body>
```

同样地，如果在 CSS 中不使用内联语法，那么可能需要使用其他的方式来实现上面的功能，如以下的代码片断：

```
<body>
    <span class="my-text" th:sytle="|color: | + ${color} + |;|">Hello World!</span>
</body>
```

以上代码片断直接使用 th:style 属性来设置 color，虽然看上去这种方式比较简单，但是如果样式很多，则会导致混乱。另外，这种方式也极大地破坏了原型页面（删除原型中的<style>节点）。

8.4.4 内联语法注释

在 JavaScript 与 CSS 中，也可以使用 Thymeleaf 的注释。注释同样分为两种：解析层注释与原型注释。只有经过模板引擎处理，解析层注释才会生效，而原型注释，只会在浏览器中打开模板源文件时生效，请见代码清单 8-28。

代码清单 8-28：codes\08\8.4\inlining\src\main\resources\comment.html

```
<head>
    <script th:inline="JavaScript">

①       /*[+
        function show() {
            alert('Hello, [[${user}]]');
        }
        +]*/

②       /*[- */
        function show() {
            alert('Prototype Content');
        }
        /* -]*/
    </script>
    <style th:inline="CSS">
③       /*[+
        .my-text{
            color: red;
        }
        +]*/

④       /*[- */
        .my-text{
            color: blue;
        }
        /* -]*/
```

```
    </style>
</head>
<body>
    <input type="button" value="click me" onClick="show()"/>
    <span class="my-text">Hello World!</span>
</body>
```

在代码清单 8-28 中既有 JavaScript 片断，也有 CSS 片断，我们先看下 JavaScript 片断（代码清单中的①和②），①中使用了“/*[+ 注释内容 +]*/”这样的注释，这是解析层注释，其在模板引擎处理前不会生效。②中使用的是原型注释：“/*[- */ 注释内容 /* -]*/”，其只会在浏览器中生效。代码清单 8-28 有一个 button，如果该模板在浏览器中打开，将会执行②的 show 方法，输出“Prototype Content”，如果通过模板引擎处理，则会执行①的 show 方法。

再看下 CSS 片断（代码清单中的③和④），③使用了解析层注释，④使用了原型注释。当这个模板页面被放到浏览器中被直接访问时，模板中节点的文字会显示为蓝色（blue），而经过模板引擎处理后，则会显示为红色（red）。注意，要想让这些注释语法生效，需要为 script 与 style 节点加上 th:inline 属性并设置 JavaScript 与 CSS 值。

除了以上的注释语法外，还可以使用“/* 表达式 */”语法来声明语句由模板处理，如以下模板内容：

```
<head>
    <script th:inline="JavaScript">
        var name = /*[[${user}]]*/ "Prototype user";
        function show() {
            alert(name);
        }
    </script>
</head>
<body>
    <input type="button" value="click me" onClick="show()"/>
</body>
```

以上代码片断中的粗体字代码表示，如果在浏览器中打开以上的代码片断，则输出“Prototype user”，经模板引擎处理后，将会输出“user”变量的值。注意，“/*[[...]]*/”表达式中的“*”与大括号间不要有空格，否则将无法识别。

8.4.5 内联语法序列化

在前面的例子中，都是使用内联语法处理字符串，其实它不仅可以处理字符串、数字、集合、Map，它甚至能处理 JavaBean，这些对象在经过处理后，在 JavaScript 中会变成 JSON 对象，请见代码清单 8-29。

代码清单 8-29：codes\08\8.4\inlining\src\main\resources\comment.html

```
<head>
    <script th:inline="JavaScript">
        // JavaBean
        var user = /*[[${dataUser}]]*/ null;
        // List
        var users = /*[[${users}]]*/ null;
        // Map
        var map = /*[[${map}]]*/ null;
        function show() {
            alert(user.name);
```

```
            alert(users.length);
            alert(map.user1.name);
        }

    </script>
</head>
<body>
    <input type="button" value="click me" onClick="show()"/>
</body>
```

在代码清单 8-29 中使用内联语法处理了 JavaBean、List 与 Map，经过模板引擎处理后，结果如下：

```
<head>
    <script>
        // JavaBean
        var user = {"id":1,"name":"Paris"};
        // List
        var users = [{"id":1,"name":"userA"},{"id":2,"name":"userB"}];
        // Map
        var map = {"user1":{"id":1,"name":"userA"},"user2":{"id":2,"name":"userB"}};
        function show() {
            alert(user.name);
            alert(users.length);
            alert(map.user1.name);
        }

    </script>
</head>
<body>
    <input type="button" value="click me" onClick="show()"/>
</body>
```

将对象转换为 JSON 的过程由 IStandardJavaScriptSerializer 接口的实现类完成，如果类路径下有 Jackson 的包，则会使用该项目进行转换，否则会使用内置的序列化类转换。在一般的环境中，会使用内置的序列化类，在 Spring Boot 中则会使用 Jackson 进行转换。

8.5 模板缓存

模板处理的性能是模板引擎的重要指标之一，改善性能的其中一个方法就使用缓存，这一节我们简单学习 Thymeleaf 的缓存机制。

8.5.1 开启与配置缓存

Thymeleaf 在处理模板时，会对模板进行解析，然后对模板内容进行处理。在处理过程中会将模板的内容以及表达式缓存起来，这样在下一次处理时，就减少了解析和处理的时间，从而提升了性能且降低了服务器的开销。下面，我们编写一个模板处理的例子，其模板内容请见代码清单 8-30。

代码清单 8-30：codes\08\8.4\inlining\src\main\resources\cacheConfig.html

```
<head>
    <script th:inline="JavaScript">
        var user = /*[[${user}]]*/ null;
    </script>
```

```
</head>
<body>
   <span th:text="|Hello World|"></span>
</body>
```

该模板中含有一段 JavaScript。下面编写处理代码，请见代码清单 8-31。

代码清单 8-31：codes\08\8.4\inlining\src\main\java\org\crazyit\boot\c8\CacheConfigTest.java

```
public class CacheConfigTest {

    public static void main(String[] args) throws Exception {
        // 模板引擎
        TemplateEngine templateEngine = new TemplateEngine();
        // 模板处理者
        ClassLoaderTemplateResolver resolover = new ClassLoaderTemplateResolver();
        // 关闭全局缓存，默认值为 true
        resolover.setCacheable(false);
        // 设置缓存仅对该页面生效
        resolover.getCacheablePatternSpec().addPattern("cacheConfig.html");
        // 缓存管理器
        StandardCacheManager cacheManager = new StandardCacheManager();
        // 设置模板缓存的最大数量
        cacheManager.setTemplateCacheMaxSize(10);
        // 设置表达式缓存的最大数量
        cacheManager.setExpressionCacheMaxSize(10);

        // 为模板引用设置解析器以及缓存管理器
        templateEngine.setTemplateResolver(resolover);
        templateEngine.setCacheManager(cacheManager);

        Context ctx = new Context();
        ctx.setVariable("user", "Angus");
        String result = templateEngine.process("cacheConfig.html", ctx);    ①
        // 该模板不会有缓存
        templateEngine.process("cacheConfig2.html", ctx);                   ②
        // 打印缓存
        printTplCache(cacheManager);
    }

    static void printTplCache(StandardCacheManager cacheManager) {
        ICache<TemplateCacheKey, TemplateModel> caches = cacheManager.getTemplateCache();
        for(TemplateCacheKey key : caches.keySet()) {
            TemplateModel model = caches.get(key);
            System.out.println(key.getTemplate() + "---" + model);
        }
    }
}
```

代码清单 8-31 中的 4 行粗体字代码对缓存进行了配置。调用 ClassLoaderTemplate- Resolver 的 setCacheable 方法关闭全局缓存，在默认情况下，全局缓存是打开的，此处之所以要关闭，是为了让本例的缓存，仅对“cacheConfig.html”模板生效。在本例中处理了两个模板（①和②）：cacheConfig.html 与 cacheConfig2.html，关闭全局缓存后，处理 cacheConfig2.html 后产生的缓存，将不会被保存。

在代码清单 8-31 的最后，会调用 printTplCache 方法，将当前的缓存输出，缓存被保存到

一个 ICache 对象中。每条缓存数据使用 key、value 的结构保存，数据的 key 是一个 TemplateCacheKey 实例，value 是一个 TemplateModel 实例。运行代码清单 8-31，控制台输出如下：

```
var user = /*[[${user}]]*/ null;
---
var user = [#th:block th:text="${user}"][/th:block];

cacheConfig.html---<head>
    <script th:inline="JavaScript">
        var user = /*[[${user}]]*/ null;
    </script>
</head>
<body>
    <span th:text="|Hello World|"></span>
</body>
```

根据结果可知，JavaScript 的内容会被当作一条缓存数据，整个模板也会被当作一条缓存数据。JavaScript 的缓存数据使用模板中<script>节点的内容作为 key，整个模板的缓存数据的 key 则是文件名。除了模板内容外，模板中的表达式也会被缓存起来，读者可以自己尝试使用 StandardCacheManager 对象来获取表达式缓存，在此不再赘述。

8.5.2 缓存有效时间

对于解析时所产生的缓存数据，有三种保存策略：

- 缓存关闭，不保存缓存数据。
- 缓存开启时不设置有效时间，缓存数据总会被保存起来，缓存的删除将由 LRU 算法（缓存淘汰算法）决定。
- 缓存开启且设置有效时间，一旦超过有效时间，在下一次处理模板时或调取缓存数据时，就会先把原来的缓存数据清除。

代码清单 8-32 可用来测试缓存的有效时间。

代码清单 8-32：codes\08\8.4\inlining\src\main\java\org\crazyit\boot\c8\CacheTest.java

```
public class CacheTest {

    public static void main(String[] args) throws Exception {
        // 模板引擎
        TemplateEngine templateEngine = new TemplateEngine();
        // 模板处理者
        ClassLoaderTemplateResolver resolover = new ClassLoaderTemplateResolver();
        // 设置缓存存活时间为 500ms
        resolover.setCacheTTLMs(500L);
        templateEngine.setTemplateResolver(resolover);
        // 获取默认的缓存管理器
        ICacheManager cacheManager = templateEngine.getCacheManager();
        Context ctx = new Context();
        String result = templateEngine.process("cache.html", ctx);
        // 输出 5 次缓存
        for(int i = 0; i < 5; i++) {
            printTplCache(cacheManager);
            // 第一次输出后，让缓存超时
```

```
            if(i == 0) {
                Thread.sleep(1000);
            }
        }
    }

    /*
     * 打印模板缓存
     */
    static void printTplCache(ICacheManager cacheManager) {
        ICache<TemplateCacheKey, TemplateModel> caches = cacheManager.getTemplateCache();
        for(TemplateCacheKey key : caches.keySet()) {
            // 缓存的 get 方法判断是否超时，是的话则删除缓存
            TemplateModel model = caches.get(key);
        }
        System.out.println("缓存数量: " + caches.keySet().size());
    }
}
```

代码清单 8-32 中的粗体字代码设置了缓存的有效时间，即在 TTL（Time to Live）属性中设置。缓存的有效时间为 500ms，在处理完 cache.html 模板后，循环 5 次输出缓存数量，第 1 次输出后睡眠 1s，让缓存超时，后面的 4 次循环，缓存的数量将为 0。运行代码清单 8-32，输出结果如下：

```
缓存数量: 1
缓存数量: 0
缓存数量: 0
缓存数量: 0
缓存数量: 0
```

一般情况下，使用默认的缓存配置即可，对于一些会动态改变的模板，我们可能需要考虑缓存的策略。

8.6 本章小结

本章深入讲解了 Thymeleaf 的模板，主要包括模板片断、模板逻辑分离、注释、内联语法与缓存等内容。本章的重点是如何使用模板片断，例如片断的定义、引用，以及片断选择器的语法，不要求熟练使用这些语法，但你看到相应的表达式时，应能知道大概的意思。后面讲解了逻辑分离与模板注释，它们的目标都是一个：让项目组内各角色协同工作，把握住这个重点，就好理解这些功能了。最后讲解了内联语法与模板缓存，不要求掌握这些功能，但在有需要时，你可以回来查阅本章内容。

CHAPTER

9

第9章 Thymeleaf 原理与扩展

本章要点

- 模板处理的方言及处理器
- 深入模板处理机制
- 模拟 Thymeleaf 模板处理逻辑
- 扩展 Thymeleaf

Thymeleaf 具有很强的扩展性，一般情况下，它提供的内置方言（Dialect）已经可以满足我们大部分的需求，在此基础上，如果有更复杂的需求，我们可以编写自己的方言，实现自己的属性、标签，就像 JSP 中的自定义标签一样。这一章，我们来讲解 Thymeleaf 的模板处理原理，了解原理后，再学习如何实现自定义的方言。通过本章的学习，你可以开发自己的模板标签、模板属性（类似“th:”属性）、内置对象，从而可以更深入地了解 Thymeleaf 的原理。

注意：本章在讲解 Thymeleaf 的模板处理机制时，会涉及设计模式、接口（对象）设计等知识，如果你对此部分内容不感兴趣，则可以直接学习本章后面关于 Thymeleaf 扩展的知识。

9.1 概念与接口介绍

本节介绍 Thymeleaf 中的一些基本概念，以及对应的接口设计，关于如何使用这些接口（对象），在下一节中讲解。

9.1.1 方言

所谓方言，我们可以理解为，它是一类语言，这类语言可以被模板引擎所识别，例如我们经常使用的 th:text 属性，它就属于 Thymeleaf 中的标准方言。同样地，如果我们有自己的方言，例如 crazy:text 这样的语句，我们要告诉模板引擎，它用来输出文本。当然我们可以编写自己的方言。

Thymeleaf 为方言定义了接口，所有的方言都要实现 org.thymeleaf.dialect.IDialect 接口，这个接口只定义了一个方法：getName。单独看这个接口，并没有特殊的含义，这个接口有多个子接口，每一个子接口在方言体系中都有具体的含义。

- IProcessorDialect：一般处理器的方言接口，该接口会提供“一般处理器”给模板引擎使用。该接口主要有 getPrefix、getDialectProcessorPrecedence 与 getProcessors 方法。getPrefix 方法返回方言的前缀，例如 Thymeleaf 内置方言的前缀是 th；getDialectProcessorPrecedence 方法返回处理器的优先级；getProcessors 方法返回处理器。
- IPreProcessorDialect：预处理器的方言接口，提供“预处理器”给模板引擎使用。该接口定义了 getDialectPreProcessorPrecedence 与 getPreProcessors 方法。
- IPostProcessorDialect：后处理器的方言接口，提供“后处理器”给模板引擎使用，该接口定义了 getDialectPostProcessorPrecedence 与 getPostProcessors 方法。
- IExpressionObjectDialect：表达式对象的方言接口，提供表达式对象的工厂对象给模板引擎使用。
- IExecutionAttributeDialect：执行属性的方言接口，提供执行属性给模板引擎使用。

Dialect 接口与以上 5 个子接口的关系及类图，请见图 9-1。

仅供非商业用途或交流学习使用

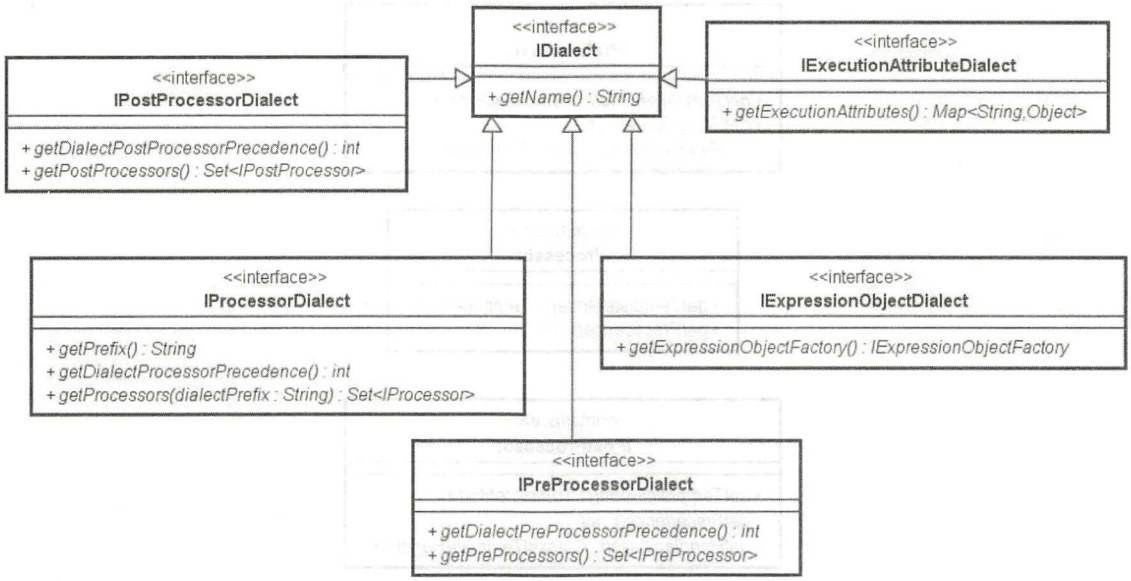


图 9-1 IDialect 接口及子接口

这 5 个子接口，大家可以理解为，它们是一类方言的 5 种形式。举个例子，将广东方言“粤语”看作一个 Dialect，在日常交际时使用方言“IProcessorDialect”，而在训斥对方时，使用方言“IPostProcessorDialect”，换言之，5 个方言子接口，是一种方言在不同场景下的表达形式。在这里，大家只需要知道方言的概念及这 5 个子接口即可，后面会详细讲解这些方言接口及相应的方法。

9.1.2 处理器

如前所述，Dialect 接口有 5 个子接口，其中 3 个是处理器的方言接口：IProcessorDialect、IPreProcessorDialect 与 IPostProcessorDialect，这 3 个方言接口都会提供处理器给模板引擎使用，分别提供以下 3 类处理器。

- **一般处理器 IProcessor**：定义了两个接口方法，分别为 getTemplateMode 与 getPrecedence，分别返回模板模式与处理器优先级。
- **预处理器 IPreProcessor**：定义了三个接口方法，分别为 getTemplateMode、getPrecedence 与 getHandlerClass，其中 getHandlerClass 方法返回一个“模板处理者”（Handler）的实例。
- **后处理器 IPostProcessor**：定义了三个接口方法，分别为 getTemplateMode、getPrecedence 与 getHandlerClass，getHandlerClass 方法同样返回一个“模板处理者”的实例。

这 3 个接口都没有父接口，3 个接口之间也没有任何关联关系，它们是各自独立存在的接口。这 3 个接口及它们的方法定义，请见图 9-2。

大家只需要知道，三个处理器方言会分别返回相应类型的处理器即可，这些处理器的使用及执行周期，会在后面讲述。

注意：Thymeleaf 的 Processor 本书称为处理器，TemplateHandler 称为模板处理者。由于处理器与处理者之间只差一个字，为避免混淆，本书在大部分情况下，都会直接使用英文 Processor 与 Handler。

仅供非商业用途或交流学习使用

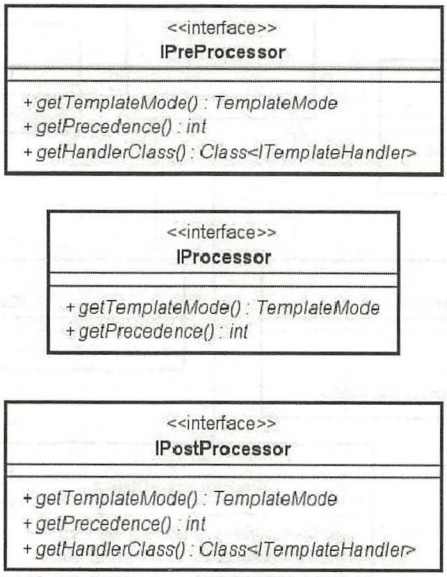


图 9-2　三大处理器接口

9.1.3　模板处理者

模板处理者对应的接口为 ITemplateHandler，整个模板的处理周期实际上都是由 Handler 来执行的。所有的 Handler 会形成一条处理链，这个执行过程，会在后面详细讲解。这里先了解一下 ITemplateHandler 接口，该接口主要有以下方法。

- setContext(ITemplateContext cxt)：设置模板引擎上下文，在处理模板时，需要获取模板引擎的相应属性。
- setNext(ITemplateHandler next)：设置处理链的下一任处理者。
- handlerXXX 方法：定义了多个 handler 方法，在遇见不同类型的模板时，可以进行不同的处理，例如遇到 HTML 的打开标签（例如），就调用 handlerOpenElement 方法，遇到关闭标签（例如）就调用 handlerCloseElement 方法。

一般情况下，ITemplateHandler 有两个具体的实现类：ProcessorTemplateHandler 与 OutputTemplateHandler，前者用于处理模板，后者主要用于输出模板，所有的一般处理器，都会在 ProcessorTemplateHandler 中执行。

9.1.4　模板解析事件

模板在处理前会进行解析，解析过程中会产生一系列的事件，例如开始解析、遇到打开标签、遇到关闭标签、遇到文本、结束解析等，对于每一个事件，最终都会产生一个 IEngineTemplateEvent 的实例。例如解析时遇到了模板节点，则这个具体的 IEngineTemplateEvent 实例表示一个事件，同时也表示一个特定模板节点的标签（实现了标签的相关接口）。IEngineTemplateEvent 接口继承于 ITemplateEvent，定义了以下方法。

- accept(IModelVisitor visitor)：在读取模板节点时，ITemplateEvent 实例也是一个访问者模式中的可访问对象，提供 accept 方法访问事件本身。
- getCol()：解析时产生事件的模板位置，该方法返回模板的列值。

仅供非商业用途或交流学习使用

- getLine()：解析时产生事件的模板位置，该方法返回模板行，行与列可定位到产生事件的具体模板位置。
- getTemplateName()：返回模板名称。
- hasLocation()：判断当前的事件是否是位置事件，如果不是位置事件，col 与 line 都是 -1，则 hasLocation 方法返回 false，而对于开始解析模板等没有模板位置的事件，该方法返回 false。
- write(Writer writer)：该方法最后会由执行输出的 Handler 调用，用于输出每个事件（或模板）的处理结果。
- beHandled(ITemplateHandler handler)：执行 Handler 的相应方法，模板被解析完后，会生成一系列的 IEngineTemplateEvent，所有的 IEngineTemplateEvent 实例都会执行这个方法进行模板处理。

图 9-3 展示了 IEngineTemplateEvent 中的 3 个实现类。

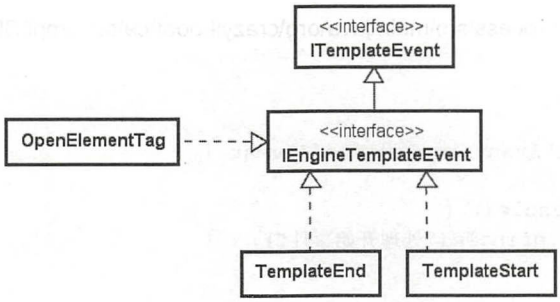


图 9-3 IEngineTemplateEvent 的部分实现类

如图 9-3 所示，TemplateStart 类与 TemplateEnd 类是单纯的事件类，而 OpenElementTag 类既表示一个事件，也表示相对应的模板节点标签。

9.1.5 节点与标签

在前面章节中，我们都将 HTML 的 Element 称为“节点”，例如“Prototype”，这是一个模板节点，th:text 被称为节点的属性。本章又出现了标签的概念，这个 span 节点含有两个标签：打开节点的标签（）与关闭节点的标签（）。要注意节点与标签这两个概念，节点是指整个 HTML 的 Element，而标签是节点的其中一部分。

9.2 深入 Thymeleaf 模板处理

这一节，我们将更加深入地讲解 Thymeleaf 的模板处理过程。这一节的内容是以 9.1 节的内容为基础的，首先要理解前面所讲述的几个概念。下面，我们先学习两个重要的设计模式。

9.2.1 策略模式

策略模式是 GoF 23 种设计模式中的一种，属于行为型模式。该模式先定义一个策略接口，而在各个实现类中封装不同的实现。使用这个模式，实现类的变化不会对客户端产生影响。以下是几个策略模式的几个参与角色。

仅供非商业用途或交流学习使用

- **策略接口**：定义公共接口，在客户端中直接调用这个接口来使用具体的策略。
- **策略实现类**：实现策略接口，策略逻辑在其里面体现。
- **客户端**：维护一个或多个策略接口，直接调用策略的接口方法，具体的策略实现，由外部决定。

下面，我们使用策略模式来编写一个简单的例子，其策略接口及策略实现类请见代码清单 9-1。

代码清单 9-1：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\stra\IEvent.java

```
public interface IEvent {

    /**
     * 策略接口定义的处理方法
     */
    void handle();
}
```

codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\stra\impl\StartEvent.java

```
/**
 * 开始实现类
 */
public class StartEvent implements IEvent {

    public void handle() {
        System.out.println("处理开始事件");
    }
}
```

codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\stra\impl\TagEvent.java

```
/**
 * 节点事件的实现类
 */
public class TagEvent implements IEvent {

    public void handle() {
        System.out.println("处理节点事件");
    }
}
```

代码清单 9-1 中的 IEvent 接口就是策略模式中的策略接口，它有两个实现类，进行最简单的控制台输出。接下来编写客户端，请见代码清单 9-2。

代码清单 9-2：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\stra\EventContext.java

```
/**
 * 策略客户端，不受策略改变的影响
 */
public class EventContext {

    // 维护多个策略接口
    private List<IEvent> events;

    public EventContext(List<IEvent> events) {
        this.events = events;
```

仅供非商业用途或交流学习使用


```
    }

    /**
     * 从外部调用的处理方法
     */
    public void process() {
        // 执行具体的策略
        for(IEvent e : events) {
            e.handle();
        }
    }
}
```

在客户端中维护了多个 IEvent 的实例，在 process 方法中调用这些 IEvent 的 handle 方法。具体的 IEvent 的实现类，客户端则不需要关心，这也是策略模式所能达到的效果，即使策略改变或增加，也不需要修改客户端代码。下面编写测试代码，请见代码清单 9-3。

代码清单 9-3：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\stra\StraTest.java

```
public class StraTest {

    public static void main(String[] args) {
        // 创建策略
        IEvent startEvent = new StartEvent();
        IEvent tagEvent = new TagEvent();
        List<IEvent> events = new ArrayList<IEvent>();
        events.add(startEvent);
        events.add(tagEvent);
        // 创建客户端，传入策略
        EventContext ctx = new EventContext(events);
        // 执行处理方法
        ctx.process();
    }
}
```

在代码清单 9-3 中创建了具体的事件（策略实现类）实例，并传入客户端，客户端再调用 process 方法。运行代码清单 9-3，控制台输出如下：

```
处理开始事件
处理节点事件
```

在上面的例子中定义了一个事件接口，这个接口在策略模式中就是策略接口。实际上这个小例子，就是在 Thymeleaf 中创建、设置、触发模板事件的浓缩版。Thymeleaf 在解析模板时会创建相应的事件，这些事件会以数组形式，被设置到一个 TemplateModel 类中（类似于上面例子中的客户端 EventContext 类）。TemplateModel 在处理模板时，就会循环调用各个事件（具体策略）的处理方法。

9.2.2 责任链模式

责任链模式也是 GoF 设计模式之一，同样也是行为型模式，它将一堆行为类串起来，形成一个处理链，对于调用责任链的客户端来说，只需要知道链的第一个处理者即可，后面有多少个处理者，客户端不需要关心。责任链模式有以下的参与者。

- **处理者（Handler）接口**：定义一个处理者接口，需要定义设置后一个处理者的方法。

仅供非商业用途或交流学习使用

- **处理者实现类**：负责处理它自己的请求，维护下一任处理者的实例，处理完后，根据具体实现，可决定是否还要交给下一个处理者。
- **客户端**：维护责任链中的第一个处理者，并向其提交处理请求。

同样地，下面我们使用责任链模式编写一个简单的例子。Handler 接口及其实现类请见代码清单 9-4。

代码清单 9-4：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\chain\IHandler.java

```
/**
 * 处理者接口
 */
public interface IHandler {

    /**
     * 设置下一任处理者
     */
    void setNext(IHandler next);

    void handlerStart();

    void handlerTag();
}
```

codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\chain\impl\HandlerA.java

```
public class HandlerA implements IHandler {

    private IHandler next;

    public void setNext(IHandler next) {
        this.next = next;
    }

    public void handlerStart() {
        // HandlerA 只实现该方法，简单打印
        System.out.println("HandlerA handlerStart 方法");
        // 交给下一任处理者
        if(next != null) {
            next.handlerStart();
        }
    }

    public void handlerTag() {
        // HandlerA 不实现该方法，直接交给下一任处理
        if(next != null) {
            next.handlerTag();
        }
    }
}
```

codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\chain\impl\HandlerB.java

```
public class HandlerB implements IHandler {

    private IHandler next;

    public void setNext(IHandler next) {
```

仅供非商业用途或交流学习使用


```
        this.next = next;
    }

    public void handlerStart() {
        // HandlerB 不实现该方法，直接交给下一任处理
        if(next != null) {
            next.handlerStart();
        }
    }

    public void handlerTag() {
        // HandlerB 实现该方法，做简单的控制台输出
        System.out.println("HandlerB handlerTag 方法");
        // 交给下一任处理
        if(next != null) {
            next.handlerTag();
        }
    }
}
```

在代码清单 9-4 中定义了 IHandler 接口，并且提供了两个实现类：HandlerA 与 HandlerB，每个实现类，都要实现接口的 setNext 方法，用来设置下一任的处理者。HandlerA 自己会维护下一任处理者的实例，它并不知道下一任处理者的实现类是谁，它也不需要关心。HandlerA 只会实现 handlerStart 方法（进行简单的控制台打印），而它不用实现 handlerTag 方法，直接让下一任处理者去实现。HandlerB 的实现与 HandlerA 类似，它会实现 handlerTag 方法，而不会实现 handlerStart 方法。

对于责任链上面的处理者来说，它们都不知道自己的上一任、下一任是谁，它们只需要把自己该做的事做好，然后再交给下一任去处理。代码清单 9-5 是责任链的客户端实现。

代码清单 9-5：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\chain\ChainClient.java

```
public class ChainClient {

    // 维护第一任处理者的实例
    private IHandler firstHandler;

    public ChainClient(IHandler firstHandler) {
        this.firstHandler = firstHandler;
    }

    /**
     * 开始执行责任链
     */
    public void doChain() {
        // 先执行开始方法
        this.firstHandler.handlerStart();
        // 再执行 tag 方法
        this.firstHandler.handlerTag();
    }
}
```

在客户端中，会维护一个 IHandler 的实例，这个处理者是整条责任链中的第一任处理者。在 doChain 方法中，会分别执行第一任处理者的两个方法，这样的话，实际上就运行了两次责任链：假如责任链上面有两个处理者，那么第一次运行责任链时，会先执行第一任处理者的

仅供非商业用途或交流学习使用

handlerStart 方法，再执行第二任处理者的 handlerStart 方法，第二次运行责任链时，会执行两个处理者的 handlerTag 方法。

测试代码请见代码清单 9-6。

代码清单 9-6：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\chain\ChainTest.java

```
public class ChainTest {

    public static void main(String[] args) {
        // 初始化责任链
        IHandler first = new HandlerA();
        IHandler second = new HandlerB();
        // 为第一个处理者设置下一任处理者
        first.setNext(second);
        // 创建客户端实例，并设置第一任处理者
        ChainClient chain = new ChainClient(first);
        // 执行责任链
        chain.doChain();
    }
}
```

该测试代码会先初始化整条责任链，HandlerA 为第一任处理者，HandlerB 为第二任处理者。设置好后，再为客户端设置第一任处理者，最后执行 doChain 方法。运行代码清单 9-6，控制台输出如下：

```
HandlerA handlerStart 方法
HandlerB handlerTag 方法
```

在责任链设计模式中，客户端代码只需要知道第一任处理者就可以了，而整条链的顺序，在初始化时决定。这个责任链的例子，实际上就是 Thymeleaf 处理模板的过程，Thymeleaf 在处理模板时，会收集“预处理器（PreProcessor）”与“后处理器（PostProcessor）”提供的处理者类（Handler），然后将这些 Handler 设置到责任链中，这样，在执行过程中，就会执行多次责任链。

Thymeleaf 的模板解析、处理过程就是策略模式结合责任链模式的过程。下面我们将这两种设计模式整合到一起来使用。

9.2.3 合并使用策略模式与责任链模式

Thymeleaf 在解析与处理模板时，使用了策略模式与责任链模式。结合这两种模式较为简单，只需要为策略接口加入相应的处理者，也就是说，为策略的客户端加入责任链的第一任处理者即可。修改前面的两个设计模式，先为策略接口加入处理者（Handler）参数，接口及实现类请见代码清单 9-7。

代码清单 9-7：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\mix\IEvent.java

```
public interface IEvent {

    /**
     * 策略接口定义的处理方法
     */
    void handle(IHandler handler);
}
```

codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\mix\impl\StartEvent.java

```
public class StartEvent implements IEvent {

    public void handle(IHandler handler) {
        System.out.println("在策略实现类中处理开始事件");
        // 执行 Handler 的 start 方法
        handler.handlerStart();
    }
}
```

codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\mix\impl\TagEvent.java

```
public class TagEvent implements IEvent {

    public void handle(IHandler handler) {
        System.out.println("在策略实现类中处理节点事件");
        handler.handlerTag();
    }
}
```

两个实现类会根据实际情况调用各自的处理方法，例如 StartEvent 调用 IHandler 的 handlerStart 方法。接下来，修改策略的客户端类，它需要维护一个责任链的处理者实例，请见代码清单 9-8。

代码清单 9-8：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\mix\EventContext.java

```
public class EventContext {

    // 维护多个策略接口
    private List<IEvent> events;

    private IHandler firstHandler;

    public EventContext(List<IEvent> events, IHandler firstHandler) {
        this.events = events;
        this.firstHandler = firstHandler;
    }

    /**
     * 从外部调用的处理方法
     */
    public void process() {
        // 执行具体的策略
        for(IEvent e : events) {
            e.handle(firstHandler);
        }
    }
}
```

在原来的 process 方法中，会循环执行当前类维护的全部策略（事件实例），在调用事件的 handler 方法时，传入责任链的第一任处理者（Handler）。假设有三个事件（A、B、C），责任链中有两个处理者（甲、乙），那么每一个事件都会执行一次责任链，执行过程如下：

```
事件 A >> 甲处理者 >> 乙处理者
事件 B >> 甲处理者 >> 乙处理者
事件 C >> 甲处理者 >> 乙处理者
```

编写测试代码，初始化事件、责任链，最后调用 EventContext 的 process 方法，请见代码清单 9-9。

代码清单 9-9：codes\09\tpl-process\src\main\java\org\crazyit\boot\c9\mix\MixTest.java

```
public class MixTest {

    public static void main(String[] args) {
        // 创建解析事件
        IEvent startEvent = new StartEvent();
        IEvent tagEvent = new TagEvent();
        List<IEvent> events = new ArrayList<IEvent>();
        events.add(startEvent);
        events.add(tagEvent);
        // 初始化责任链
        IHandler handlerA = new HandlerA();
        IHandler handlerB = new HandlerB();
        handlerA.setNext(handlerB);
        // 执行责任链
        EventContext ctx = new EventContext(events, handlerA);
        ctx.process();
    }
}
```

运行代码清单 9-9，输出结果如下：

```
在策略实现类中处理开始事件
HandlerA handlerStart 方法
在策略实现类中处理节点事件
HandlerB handlerTag 方法
```

在测试代码中设置了两个事件、两个处理者，它们的执行顺序如下：

```
StartEvent.handler >> HandlerA.handlerStart >> HandlerB.handlerStart
TagEvent.handler >> HandlerA.handlerTag >> HandlerB.handlerTag
```

以上例子将策略模式和责任链模式结合到一起使用，其逻辑非常简单，即在策略接口中调用责任链的第一任处理者，这样做的好处是，后面如果还需要添加事件实现类，或者添加责任链的处理者，都不需要修改客户端的代码，但是需要修改初始化事件、初始化责任链的逻辑。想更清晰地了解整个过程的参与对象，可以参看图 9-4。

再次说明，本例的关键在于 EventContext 类，它维护一个事件集合和一个处理者。在执行 process 方法时，会遍历事件集合并调用事件的 handle 方法，并且会将责任链的第一个处理者传入，换言之，每个事件都会执行整条责任链。在 Thymeleaf 中，由 TemplateModel 类来完成 EventContext 的工作。

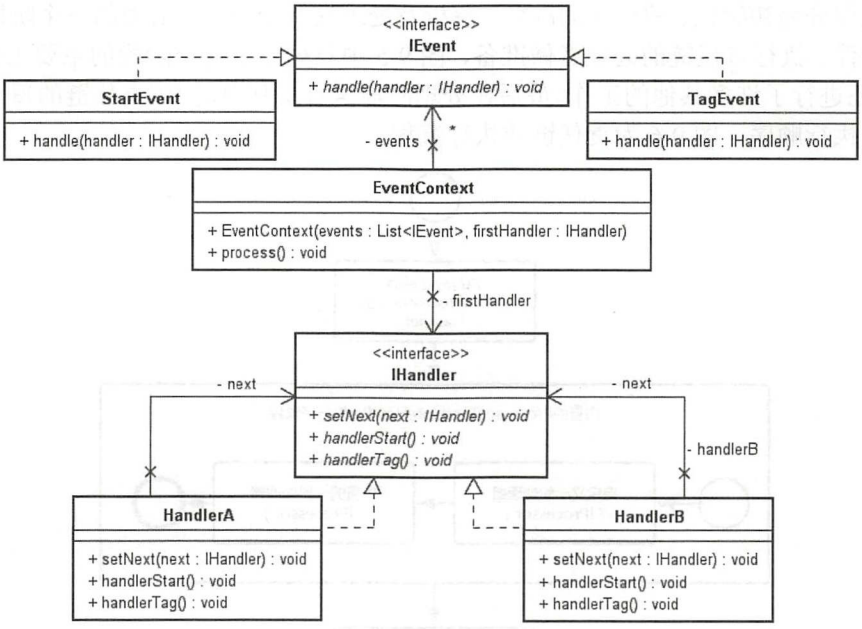


图 9-4 本例的类图

9.2.4 模板处理过程

Thymeleaf 在处理模板的过程中，会结合使用策略模式与责任链模式。在解析模板时，会产生多个模板事件，然后会遍历这些事件，每一个事件都会执行整条责任链。模板引擎处理模板的大概流程请见图 9-5。

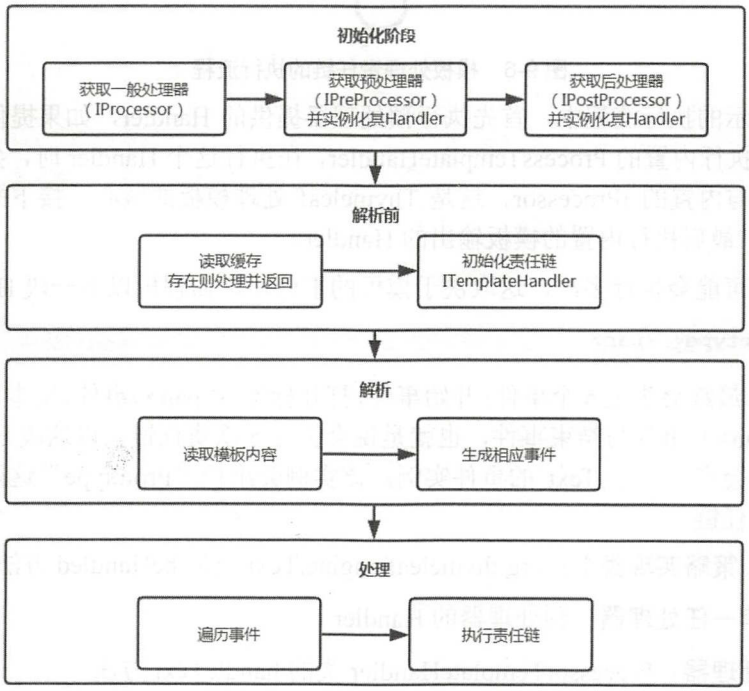


图 9-5 模板处理过程

整个过程分为初始化、解析前的准备、解析及处理这几个阶段，前面的三个阶段都是为了最后遍历事件、执行责任链的处理者做准备，图 9-5 中只列出了各个阶段的主要工作，实际上 Thymeleaf 还进行了许多其他的工作。最后在处理阶段执行责任链时，该责任链的每个处理者，都有相应的执行顺序，图 9-6 为责任链的执行流程。

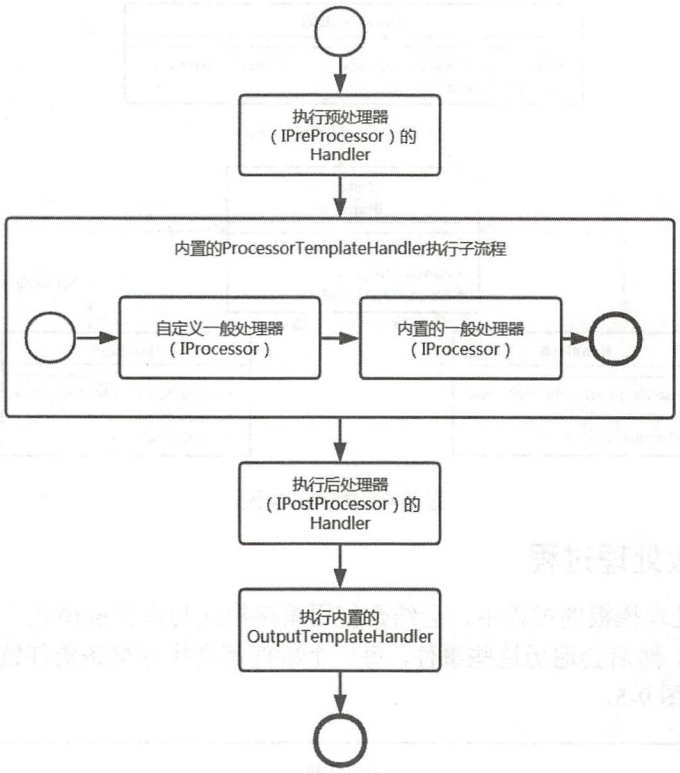


图 9-6　模板处理责任链的执行流程

在图 9-6 所示的执行流程中，首先执行预处理器提供的 Handler，如果提供多个，就按顺序执行。然后再执行内置的 ProcessTemplateHandler，在执行这个 Handler 时，会执行我们自定义的 IProcessor 与内置的 IProcessor，这是 Thymeleaf 处理模板的核心。接下来执行后处理器提供的 Handler，最后执行内置的模板输出的 Handler。

整个责任链可能会执行多次，这取决于模板的事件，例如解析以下一段 HTML：

```
<span>Prototype</span>
```

这段 HTML 最终会生成 5 个事件：开始事件、打开标签（）事件、文本事件（Prototype）、关闭标签（）事件与结束事件，也就是说会执行 5 次责任链。以这段 HTML 产生的文本事件为例，它会产生一个 Text 的事件实例，该实例会维护“Prototype”这段内容，然后按以下顺序执行责任链。

- **事件类（策略实现类）：** org.thymeleaf.engine.Text 类的 beHandled 方法
- **责任链第一任处理器：** 预处理器的 Handler
- **责任链处理器：** ProcessorTemplateHandler 类的 handleText 方法
- **责任链处理器：** 后处理器的 Handler

- 责任链最后一个处理器：OutputTemplateHandler 类的 handleText 方法

其中，ProcessorTemplateHandler 的 handleText 方法的执行过程为，如果有自定义的“一般处理器（IProcessor）”，会先执行该处理器，然后再执行内置的一般处理器。除了文本事件外，其他的事件也有类似的执行过程。下面介绍 Thymeleaf 的事件。

9.2.5 模板事件

Thymeleaf 在解析模板时会产生一系列的事件，这些事件都是 IEngineTemplateEvent 接口的实现类，这些事件的 Java 类都放在“org.tyhmeleaf.engine”包下面。主要有以下几个事件的实现类。

- TemplateStart：开始解析时产生的事件，只是单纯的开始事件。
- TemplateEnd：结束解析时产生的事件，只是单纯的结束事件。
- CDATASection：解析模板遇到“<![CDATA[...]]>”标签时产生的事件。
- Comment：解析模板遇到 HTML 注释“<!--...-->”时产生的事件。
- Text：解析模板遇到普通文本时产生的事件。
- XMLDeclaration：解析 XML 声明时产生的事件，例如模板中含有“<?xml version="1.0"?>”，则会产生该事件。
- DocType：解析“<!doctype html>”语句时产生的事件。
- ProcessingInstruction：解析 XML 文档中的处理指令时产生的事件，例如“<?xml-stylesheet ?>”语句。
- OpenElementTag：遇到打开标签时产生的事件，例如“”。
- CloseElementTag：遇到关闭标签时产生的事件，例如“”。
- StandaloneElementTag：遇到标准标签时产生的事件，例如“”。在解析时如果产生了 StandaloneElementTag 事件，其后就不会再产生 OpenElementTag 与 CloseElementTag 事件了。

解析产生这些事件对象后，就会调用它们的 beHandled 方法（IEngineTemplateEvent 接口方法）来执行责任链的第一任处理器，每一个事件都会执行一次责任链。

这一节深入讲解了 Thymeleaf 的模板处理机制，要想了解整个处理过程，只要理解策略与责任链两个设计模式即可。Thymeleaf 以这两个设计模式为基础，添加模板事件、责任链的处理者等对象，从而实现模板解析、处理等功能。

9.3 处理器

Thymeleaf 里面有 3 类处理器：预处理器（IPreProcessor）、一般处理器（IProcessor）与后处理器（IPostProcessor），本节将讲解如何使用这些处理器。

9.3.1 预处理器

一个自定义的预处理器需要实现 IPreProcessor 接口，根据前面章节的介绍，预处理器提供的 Handler 类将会被放到责任链中执行，如果只有一个预处理器，那么这个 Handler 将会作

为责任链的第一任处理者。在代码清单 9-10 中编写了一个自定义的预处理方言类及两个预处理器。

代码清单 9-10：codes\09\processor\src\main\java\org\crazyit\boot\c9\pre\MyPreDialect.java

```
public class MyPreDialect implements IPreProcessorDialect {

    public String getName() {
        return "Pre Dialect";
    }

    public int getDialectPreProcessorPrecedence() {
        return 0;
    }

    public Set<IPreProcessor> getPreProcessors() {
        // 添加两个预处理器
        Set<IPreProcessor> ps = new HashSet<IPreProcessor>();
        ps.add(new PreProcessorA());
        ps.add(new PreProcessorB());
        return ps;
    }
}
```

codes\09\processor\src\main\java\org\crazyit\boot\c9\pre\processor\PreProcessorA.java

```
public class PreProcessorA implements IPreProcessor {

    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    public int getPrecedence() {
        return 0;
    }

    public Class<? extends ITemplateHandler> getHandlerClass() {
        return PreHandlerA.class;
    }
}
```

codes\09\processor\src\main\java\org\crazyit\boot\c9\pre\processor\PreProcessorB.java

```
public class PreProcessorB implements IPreProcessor {

    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    public int getPrecedence() {
        return 0;
    }

    public Class<? extends ITemplateHandler> getHandlerClass() {
        return PreHandlerB.class;
    }
}
```

代码清单 9-10 中的两个预处理器直接实现 IPreProcessor 接口，它们分别指定了

PreHandlerA 与 PreHandlerB 两个处理者（Handler），这两个处理者将会被放到责任链中。处理者的实现非常简单，为节省篇幅，此处仅贴出 PreHandlerA 的实现，请见代码清单 9-11。

代码清单 9-11：codes\09\processor\src\main\java\org\crazyit\boot\c9\pre\handler\PreHandlerA.java

```
public class PreHandlerA extends AbstractTemplateHandler {

    @Override
    public void handleText(IText text) {
        System.out.println("PreHandlerA 处理的文本: " + text + " 下一任处理者: " +
this.getNext());
        // 让责任链中的下一任处理者去执行
        super.handleText(text);
    }
}
```

PreHandlerA 处理者仅实现了 handleText 方法，该方法会简单地输出字符串与下一任处理者的实例，如果处理的模板中有文本，就会经过 PreHandlerA 的 handleText 方法输出。本例的模板非常简单，内容为“Angus”。编写测试代码，请见代码清单 9-12。

代码清单 9-12：codes\09\processor\src\main\java\org\crazyit\boot\c9\pre\PreTest.java

```
public class PreTest {

    public static void main(String[] args) {
        // 创建模板引擎
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        // 添加自定义的方言
        templateEngine.addDialect(new MyPreDialect());
        Context ctx = new Context();
        String result = templateEngine.process("pre/index.html", ctx);
        System.out.println(result);
    }
}
```

运行代码清单 9-12，控制台输出如下：

```
    PreHandlerA 处理的文本: Angus 下一任处理者:
org.crazyit.boot.c9.pre.handler.PreHandlerB@11c7bf6
    PreHandlerB 处理的文本: Angus 下一任处理者:
org.thymeleaf.engine.ProcessorTemplateHandler@1b485ba
    <span>Angus</span>
```

根据输出结果可知，“Angus”字符串会被两个 Handler 处理，PreHandlerA 的下一任处理者是 PreHandlerB，而 PreHandlerB 的下一任处理者就是系统内置的 ProcessorTemplateHandler。

以上例子在解析“Angus”模板时，会生成 5 个事件，按顺序分别为：开始事件、打开标签事件、文本事件、关闭标签事件与结束事件。如果想让 PreHandlerA 处理器处理其他事件，则可以重写父类的其他方法，例如要处理模板开始解析的方法，可重写 handleTemplateStart 方法。Handler 需要处理什么，就重写什么方法。只要知道整个模板的处理原理，就会知道，每一个事件都会执行完整的一条责任链。

9.3.2 后处理器

后处理器与预处理器类似，它需要实现 IPostProcessor 接口。我们在这里再复习一下

Handler 责任链的执行流程，请见图 9-7。

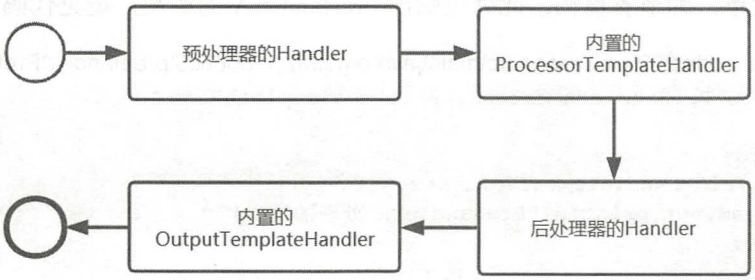


图 9-7 Handler 责任链

图 9-7 是图 9-6 的缩略版，目的是让大家了解后处理器所处的位置。在责任链中，我们提供的后处理器将会是系统内置的 ProcessorTemplateHandler 的下一任。代码清单 9-13 实现了一个后处理器。

代码清单 9-13：codes\09\processor\src\main\java\org\crazyit\boot\c9\post\process\MyPostProcessor.java

```
public class MyPostProcessor implements IPostProcessor {

    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    public int getPrecedence() {
        return 0;
    }

    public Class<? extends ITemplateHandler> getHandlerClass() {
        return PostHandler.class;
    }
}
```

与前面的预处理器类似，要先创建 IPostProcessorDialect 的实现类，再设置代码清单 9-13 中的自定义后处理器。方言类、处理者类与前面的预处理器类似，在此不再赘述，大家只要注意方言类接口是 IPostProcessorDialect、处理器接口是 IPostProcessor 就可以了。

9.3.3 一般处理器

在责任链中，一般处理器由 Thymeleaf 内置的 ProcessTemplateHandler 执行。这个内置的 Handler，除了会执行我们自定义的一般处理器外，还会执行 Thymeleaf 内置的处理器。Thymeleaf 默认的处理，都由 ProcessTemplateHandler 中的一般处理器去执行，例如一个带有“th:text”属性的节点，最终会由 StandardTextTagProcessor 类进行处理。下面编写一个自定义的一般处理器，对文本进行处理，请见代码清单 9-14。

代码清单 9-14：codes\09\processor\src\main\java\org\crazyit\boot\c9\normal\MyProcessor.java

```
public class MyProcessor implements ITextProcessor {

    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    public int getPrecedence() {
```

```
        return 0;
    }

    public void process(ITemplateContext context, IText text,
            ITextStructureHandler structureHandler) {
        // 将字符串替换为 abc
        if(text != null && !text.getText().trim().equals("")) {
            structureHandler.setText("abc");
        }
    }
}
```

代码清单 9-14 是一个简单的一般处理器，它实现的是 ITextProcessor 接口，这是 IProcessor 的子接口，该接口专门用于处理模板中的普通文本。IProcessor 并没有定义处理方法，要处理具体的数据，需要实现相应的接口。这里会将所有待处理的文本全部设置为“abc”。使用自定义的一般处理器，同样需要建立一个方言类（IProcessorDialect 的实现类），该类与预处理器、后处理器的方言类大致相同。测试代码与模板内容请见代码清单 9-15。

代码清单 9-15：codes\09\processor\src\main\java\org\crazyit\boot\c9\normal\NormalTest.java

```
public class NormalTest {

    public static void main(String[] args) {
        // 创建模板引擎
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        // 添加自定义的方言
        templateEngine.addDialect(new MyProcessorDialect());
        Context ctx = new Context();
        String result = templateEngine.process("normal/index.html", ctx);
        System.out.println(result);
    }
}
```

codes\09\processor\src\main\resources\normal\index.html

```
<span>Angus</span>
<div>Paris</div>
```

运行代码清单 9-15，输出结果如下：

```
<span>abc</span>
<div>abc</div>
```

除了上面的文本处理器外，还有节点处理器、边界处理器，它们都属于一般处理器。再次提醒，一般处理器都由 ProcessTemplateHandler 执行，而这个 Handler 是责任链中的其中一个处理者。

9.3.4 处理器优先级

三个处理器方言接口（IxxxDialect）可以设置方言优先级，对应的处理器（IxxxProcesso）可以设置处理器优先级。不管是 IxxxDialect 接口还是 IxxxProcessor 接口，都可以实现 getxxxPrecedence 方法来设置优先级，该方法返回的 int 值越小，优先级越高。

经笔者测试，预处理器方言（IPreProcessDialect）与后处理器方言（IPostProcessDialect）这两个接口，设置的方言优先级不会产生效果。要想设置预处理器和后处理器的执行顺序，可

以直接通过设置处理器（IPreProcessor 或 IPostProcessor）的优先级来实现。

优先级只会在同类型的方言（处理器）间生效，预处理器和后处理器之间不存在优先级问题，因为在责任链中，预处理器的 Handler 总是先执行的。下面通过几张图来展示三大处理器的优先级，如图 9-8 所示。

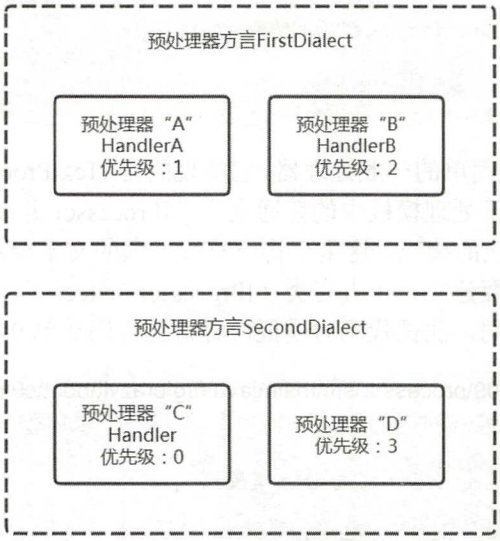


图 9-8 预处理器的优先级

图 9-8 中有两个预处理器方言，每个各有两个预处理器，预处理器 A 返回 HandlerA 处理者，其优先级为 1，其他的预处理器返回相应的处理者（Handler）与优先级，预处理器 C 的优先级是 0，因此四个 Handler 的执行顺序为：C >> A >> B >> D。后处理器的优先级与预处理器类似，在此不再赘述。下面我们来了解下一般处理器的优先级规则，请见图 9-9。

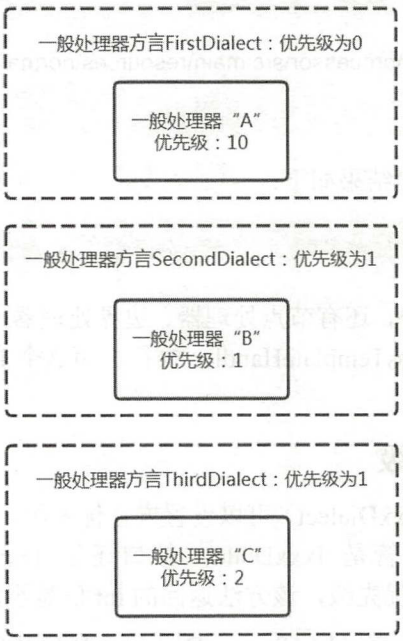


图 9-9 一般处理器优先级

与预处理器不同，一般处理器会受到处理器方言的优先级限制，如图 9-9 所示，FirstDialect 方言的优先级为 0，即使它的处理器优先级为 10，它依然会最先处理。SecondDialect 与 ThirdDialect 的方言优先级一致，但 SecondDialect 的处理器 B 的优先级为 1，因此处理器 B 会优先执行，故图 9-9 中的执行顺序为：A >> B >> C。

9.3.5 标签处理器

前面我们在一个例子中使用了文本处理器 ITextProcessor，下面我们来看看节点处理器如何使用。它也属于一般处理器，会由 ProcessTemplateHandler 执行。节点处理器对应的接口为 IElementProcessor，它有两个子接口：IElementTagProcessor 与 IElementModelProcessor，本书将 IElementTagProcessor 称为标签处理器，而将 IElementModelProcessor 称为模型处理器。代码清单 9-16 即实现了一个标签处理器。

代码清单 9-16：codes\09\processor\src\main\java\org\crazyit\boot\c9\ele\MyTagProcessor.java

```
public class MyTagProcessor implements IElementTagProcessor {

    /**
     * 返回匹配的节点名称，例如返回 div，只处理模板中的 div 节点
     */
    public MatchingElementName getMatchingElementName() {
        ElementName eleName = ElementNames.forHTMLName("div");
        MatchingElementName mEleName = MatchingElementName.forElementName(
                getTemplateMode(), eleName);
        return mEleName;
    }

    /**
     * 根据属性匹配节点，含有 crazy:test 属性的节点都会被匹配到
     */
    public MatchingAttributeName getMatchingAttributeName() {
        // 过滤节点，含有 crazy:test 前缀的节点才会被获取
        AttributeName attName = AttributeNames.forHTMLName("crazy", "test");
        MatchingAttributeName mAttName = MatchingAttributeName
                .forAttributeName(getTemplateMode(), attName);
        return mAttName;
    }

    /**
     * 返回模板模式
     */
    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    /**
     * 返回处理器优先级
     */
    public int getPrecedence() {
        return 0;
    }

    public void process(ITemplateContext context, IProcessableElementTag tag,
            IElementTagStructureHandler structureHandler) {
```

```
            System.out.println(tag.getClass());
        }
    }
```

代码清单 9-16 实现了一个简单的标签处理器，getTemplateMode 与 getPrecedence 方法是 IProcessor 接口定义的方法，分别返回模式与优先级。我们先看下两个 getMatchingXXX 方法，这两个方法主要用来匹配模板中的节点，例如本例中的 getMatchingElementName 方法只处理模板中的 div 节点，getMatchingAttributeName 方法只处理含有“crazy:test”属性的节点，将这两个方法相结合，则只处理含有 crazy:test 属性的 div 节点。

标签处理器的 process 方法，会将参数 IProcessableElementTag 对象的 class 输出，该对象既是一个标签对象，也是一个模板事件。让模板引擎解析以下的模板：

```
<div crazy:test="Paris">Angus</div>
<div crazy:test="Angus"/>
```

输出结果如下：

```
class org.thymeleaf.engine.OpenElementTag
class org.thymeleaf.engine.StandaloneElementTag
```

根据结果可知，如果模板标签是一个标准标签（ ），则 process 方法得到的是一个 StandaloneElementTag 实例，如果是一个普通标签（ ），就返回一个 OpenElementTag 实例。有多少个模板节点，就会执行多少次process方法，该方法拿到的只会是StandaloneElementTag 或 OpenElementTag 实例。

process 方法拿到相应的标签对象后，我们就可以使用 process 方法参数中的 IElementTagStructureHandler 对象操作模板节点，该对象主要有以下功能。

- **属性处理**：可调用 setAttribute、replaceAttribute、removeAttribute 等方法操作节点的属性。
- **设置节点体**：可调用 setBody 方法设置节点内容。
- **移除节点内容**：可调用 removeElement、removeTags 等方法删除节点或节点内容。
- **循环节点**：使用 iterateElement 方法，传入集合数据，循环当前节点。

可见，IElementTagStructureHandler 的方法都是用于操作当前节点的。在本章后面讲述扩展时，我们会使用到这些方法，在这里，大家对这个接口的功能有个初步了解即可。

9.3.6　模型处理器

模型处理器对应的接口是 IElementModelProcessor，它也属于节点处理器的一种。例如，在代码清单 9-17 中即实现了一个模型处理器。

代码清单 9-17：codes\09\processor\src\main\java\org\crazyit\boot\c9\model\MyModelProcessor.java

```
/**
 * 自定义模型处理器
 */
public class MyModelProcessor implements IElementModelProcessor {

    /**
     * 两个节点匹配方法返回 null，表示将会处理所有节点
     */
```

```
    public MatchingElementName getMatchingElementName() {
        return null;
    }

    public MatchingAttributeName getMatchingAttributeName() {
        return null;
    }

    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    public int getPrecedence() {
        return 0;
    }

    public void process(ITemplateContext context, IModel model,
            IElementModelStructureHandler structureHandler) {
        for (int i = 0; i < model.size(); i++) {
            System.out.println(model.get(i).getClass().getName() + ": "
                    + model.get(i));
        }
    }
}
```

模型处理器与标签处理器类似，都需要实现两个节点匹配方法，以及 process 方法。这里的两个节点匹配方法返回 null，表示模板中所有的节点都会被匹配到。process 方法会将 IModel 中维护的所有 Tag 对象输出，并将下面这段 HTML 提供给模板引擎：

```
<div>Angus</div>
```

输出结果如下：

```
org.thymeleaf.engine.OpenElementTag: <div>
org.thymeleaf.engine.Text: Angus
org.thymeleaf.engine.CloseElementTag: </div>
```

可以看到，IModel 里面保存了该节点中的所有组成部分，并将它们封装为相应的类。如果存在多个节点，那么 process 方法将会执行多次，并将以下的代码交给模板引擎处理：

```
<div><span>Angus</span></div>
```

输出结果如下：

```
org.thymeleaf.engine.OpenElementTag: <div>
org.thymeleaf.engine.OpenElementTag: <span>
org.thymeleaf.engine.Text: Angus
org.thymeleaf.engine.CloseElementTag: </span>
org.thymeleaf.engine.CloseElementTag: </div>
org.thymeleaf.engine.OpenElementTag: <span>
org.thymeleaf.engine.Text: Angus
org.thymeleaf.engine.CloseElementTag: </span>
```

根据结果可知，处理 div 时会执行一次 process 方法，执行 div 下面的 span 节点时，也会执行一次 process 方法。

模型处理器与标签处理器的区别是，标签处理器只能拿到当前节点的标签对象，而模型处

理器则可以拿到整个节点模型。

9.3.7 边界处理器

边界处理器也属于一般处理器，同样由 ProcessorTemplateHandler 类执行。边界处理器提供两个方法，这两个方法会在模板开始处理前、处理完成后执行，在开始方法与结束方法之间，会执行全部的“一般处理器”。例如，在代码清单 9-18 中即实现了一个简单的边界处理器。

代码清单 9-18：codes\09\processor\src\main\java\org\crazyit\boot\c9\bondary\MyBoundariesProcessor.java

```
public class MyBoundariesProcessor implements ITemplateBoundariesProcessor {

    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    public int getPrecedence() {
        return 0;
    }

    /**
     * 该方法将会在所有的一般处理器前面执行
     */
    public void processTemplateStart(ITemplateContext context,
            ITemplateStart templateStart,
            ITemplateBoundariesStructureHandler structureHandler) {
        System.out.println("===========  start");
    }

    /**
     * 该方法将会在所有的一般处理器后面执行
     */
    public void processTemplateEnd(ITemplateContext context,
            ITemplateEnd templateEnd,
            ITemplateBoundariesStructureHandler structureHandler) {
        System.out.println("===========  end");
    }
}
```

对于边界处理器，要注意两点：第一，要与预处理器和后处理器区分开，预处理器总会在一般处理器前执行，后处理器总会在一般处理器后执行；第二，对于一个模板，开始方法与结束方法只会执行一次，不管里面有多少个节点。

9.3.8 其他处理器

前面讲解了节点处理器（标签、模型）与边界处理器。如果模板中存在其他类型的文本，我们也可以自定义相应的处理器。我们可以使用以下的一般处理器接口。

- ITextProcessor：文本处理器接口。
- ICommentProcessor：注释处理器接口。
- ICDATASectionProcessor：CDATA 标签的处理器接口。
- IDocTypeProcessor：“<!doctype html>”标签的处理器接口。
- IXMLDeclarationProcessor：XML 声明的处理器接口。

仅供非商业用途或交流学习使用

➢ IProcessingInstructionProcessor：XML 指令的处理器接口。

总之，想要处理什么数据，就实现什么接口。以上接口都有 process 方法，你可以通过该方法拿到想要处理的数据。这些处理器的实现，可参照前面介绍的几个处理器，在此不再赘述。

9.4 Thymeleaf 扩展

我们在本章的前面章节中讲解了 Thymeleaf 的工作机制，如果你还没有掌握模板的处理过程，也不影响本节内容的学习，但笔者建议，起码看一下本章前面的 Thymeleaf 模板处理流程图，大概了解一下模板的处理过程，以便更好地掌握本节内容。

9.4.1 自定义标签

实现自定义标签非常简单，只要实现一个标签处理器即可。下面，我们在 Thymeleaf 下实现一个常用的 JSTL 标签：<c:if>。该标签处理器的实现请见代码清单 9-19。

代码清单 9-19：codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\tag\MyTagProcessor.java

```
public class MyTagProcessor implements IElementTagProcessor {

    /**
     * 返回匹配的节点名称
     */
    public MatchingElementName getMatchingElementName() {
        ElementName eleName = ElementNames.forHTMLName("c:if");
        MatchingElementName mEleName = MatchingElementName.forElementName(
                getTemplateMode(), eleName);
        return mEleName;
    }

    /**
     * 根据属性匹配节点，含有 test 属性的节点都会被匹配到
     */
    public MatchingAttributeName getMatchingAttributeName() {
        AttributeName attName = AttributeNames.forHTMLName("", "test");
        MatchingAttributeName mAttName = MatchingAttributeName
                .forAttributeName(getTemplateMode(), attName);
        return mAttName;
    }

    /**
     * 返回模板模式
     */
    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    /**
     * 返回处理器优先级
     */
    public int getPrecedence() {
        return 0;
    }

    /**
```


仅供非商业用途或交流学习使用

```
     * 处理<c:if test>的节点
     */
    public void process(ITemplateContext context, IProcessableElementTag tag,
            IElementTagStructureHandler structureHandler) {
        // 创建表达式解析器
        IStandardExpressionParser expressionParser = StandardExpressions
                .getExpressionParser(context.getConfiguration());
        // 获取属性值
        String expressionValue = tag.getAttributeValue("test");
        // 解析表达式
        final IStandardExpression expression = expressionParser
                .parseExpression(context, expressionValue);
        // 执行表达式
        final Object value = expression.execute(context);
        // 将结果转换为 boolean 值
        boolean result = EvaluationUtils.evaluateAsBoolean(value);
        // 处理节点
        if (result) {
            // 结果为 true，仅仅删除标签
            structureHandler.removeTags();
        } else {
            // 结果为 false，删除整个节点
            structureHandler.removeElement();
        }
    }
}
```

我们先看下两个节点匹配方法，getMatchingElementName 返回 c:if，getMatchingAttributeName 返回 test，也就是说，当前的处理器只会处理<c:if>这样的标签。

在 process 方法中，会先获取标签的 test 属性值，然后调用 Thymeleaf 的表达式接口来计算该表达式。除了可以使用 Thyemelaf 的 API 来计算表达式外，我们还可以使用其他的类库来解析表达式，此处为了简单起见，直接使用 Thymeleaf 的 API。计算出表达式的结果后，再判断结果，如果结果为 true，则将<c:if>标签删除，如果结果为 false，则将整个节点删除，这个节点就不会在结果中显示了。

新建一个模板，由模板引擎处理，请见代码清单 9-20。

代码清单 9-20：codes\09\custom-tpl\src\main\resources\tag\index.html

```
<c:if test="${man == true}">
    <span>男</span>
</c:if>
<c:if test="${man == false}">
    <span>女</span>
</c:if>
```

模板中的表达式会根据 boolean 变量 man 的值，来决定输出的 span 节点。运行类请见代码清单 9-21。

代码清单 9-21：codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\tag\TagTest.java

```
public class TagTest {

    public static void main(String[] args) {
        // 创建模板引擎
```


仅供非商业用途或交流学习使用

```
        TemplateEngine templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(new ClassLoaderTemplateResolver());
        // 设置自定义方言
        templateEngine.addDialect(new TagDialect());
        Context ctx = new Context();
        ctx.setVariable("man", false);
        String result = templateEngine.process("tag/index.html", ctx);
        System.out.println(result);
    }
}
```

运行代码清单 9-21，输出结果如下：

```
<span>女</span>
```

根据结果可知，我们的标签处理器已经生效。由于调用了删除节点与删除标签的方法，因此在结果中会出现很多空格，解决此问题非常简单，可以增加一个 ITextProcessor 的文本处理器，再设置这个处理器的优先级低于标签处理器的优先级（getPrecedence 方法的返回值小的先执行）。在文本处理器中，调用 trim()方法删掉文本中的空格即可。本例对应的文本处理器为 codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\tag\MyTextProcessor.java，读者可自行查看。

9.4.2 自定义模板属性

自定义模板属性与自定义标签类似，也是实现一个标签处理器。下面建立一个简单的属性，用来格式化日期，请见代码清单 9-22。

代码清单 9-22：codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\attr\AttrProcessor.java

```
public class AttrProcessor implements IElementTagProcessor {

    /**
     * 返回匹配的节点名称
     */
    public MatchingElementName getMatchingElementName() {
        return null;
    }

    /**
     * 根据属性匹配节点，含有 test 属性的节点都会被匹配到
     */
    public MatchingAttributeName getMatchingAttributeName() {
        AttributeName attName = AttributeNames.forHTMLName("crazy", "format");
        MatchingAttributeName mAttName = MatchingAttributeName
                .forAttributeName(getTemplateMode(), attName);
        return mAttName;
    }

    /**
     * 返回模板模式
     */
    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    /**
```


仅供非商业用途或交流学习使用

```
        * 返回处理器优先级
        */
       public int getPrecedence() {
           return 0;
       }

       /**
        * 处理<c:if test>的节点
        */
       public void process(ITemplateContext context, IProcessableElementTag tag,
               IElementTagStructureHandler structureHandler) {
           // 创建表达式解析器
           IStandardExpressionParser expressionParser = StandardExpressions
                   .getExpressionParser(context.getConfiguration());
           // 获取属性值
           String expressionValue = tag.getAttributeValue("crazy:format");
           // 解析表达式
           final IStandardExpression expression = expressionParser
                   .parseExpression(context, expressionValue);
           // 执行表达式
           final Object value = expression.execute(context);
           if(value instanceof Date) {
               // 转换为 Date
               Date date = (Date)value;
               // 设置 body
               String dateStr = formatDate(date);
               // 设置节点体，设置为 false 表示文本不需要再由 Thymeleaf 处理
               structureHandler.setBody(dateStr, false);
               // 移除 crazy:format 属性
               structureHandler.removeAttribute("crazy:format");
           }
       }

       // 日期格式
       private SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

       private String formatDate(Date date) {
           try {
               return sdf.format(date);
           } catch (Exception e) {
               return date.toString();
           }
       }
   }
```

该处理器中的 process 方法同样使用 Thymeleaf 的 API 来处理，解析得到 Date 实例后，再将其格式化为“yyyy-MM-dd HH:mm:ss”的形式，最后使用 IElementTagStructureHandler 来设置节点体，并删除 crazy:format 属性。本例处理的模板内容与测试代码请见代码清单 9-23。

代码清单 9-23：codes\09\custom-tpl\src\main\resources\attr\index.html

```
<span crazy:format="${date}"></span>
```

codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\attr\AttrTest.java

```
        Context ctx = new Context();
        Date now = new Date();
```


仅供非商业用途或交流学习使用

```
        ctx.setVariable("date", now);
        String result = templateEngine.process("attr/index.html", ctx);
```

运行测试代码，输出结果如下：

```
<span>2018-03-02 10:54:42</span>
```

9.4.3 自定义内置对象

Thymeleaf 提供了 5 种方言，即三大处理器方言加上表达式对象方言与执行属性方言。三大处理器方言已经在前面介绍过，本小节来介绍表达式对象方言，并且实现一个简单的内置对象。表达式对象方言实现类以及对象工厂请见代码清单 9-24。

代码清单 9-24：codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\obj\MyExpDialect.java

```
/**
 * 自定义表达式对象方言类
 */
public class MyExpDialect implements IExpressionObjectDialect {

    public String getName() {
        return "My Exp Dialect";
    }

    /**
     * 返回对象工厂实例
     */
    public IExpressionObjectFactory getExpressionObjectFactory() {
        return new MyObjectFactory();
    }
}
```

codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\obj\MyObjectFactory.java

```
public class MyObjectFactory implements IExpressionObjectFactory {

    /**
     * 返回表达式对象的名称
     */
    public Set<String> getAllExpressionObjectNames() {
        Set<String> names = new HashSet<String>();
        names.add("crazy");
        return names;
    }

    /**
     * 创建内置对象对应的实例
     */
    public Object buildObject(IExpressionContext context,
            String expressionObjectName) {
        if("crazy".equals(expressionObjectName)) {
            return new CrazyUtil();
        }
        return null;
    }

    /**
     * 是否缓存该对象
```


仅供非商业用途或交流学习使用

```
     */
    public boolean isCacheable(String expressionObjectName) {
        return false;
    }
}
```

我们主要看下对象工厂类，getAllExpressionObjectNames 方法会直接返回表达式对象的名称，也就是内置对象的名称，buildObject 方法返回“crazy”对应的实例，那么在 Thymeleaf 的表达式中，可以使用“#crazy”这样的语句来调用 CrazyUtil 类的方法，就像在前面章节中使用“#context”、“#strings”一样。CrazyUtil 是一个简单的工具类，其源码请见代码清单 9-25。

代码清单 9-25：codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\obj\CrazyUtil.java

```
public class CrazyUtil {

    /**
     * 添加天数
     */
    public String addDay(String dateString, int day) {
        try {
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
            Calendar c = Calendar.getInstance();
            c.setTime(sdf.parse(dateString));
            c.add(Calendar.DAY_OF_MONTH, day);
            return sdf.format(c.getTime());

        } catch (Exception e) {
            e.printStackTrace();
            return dateString;
        }
    }
}
```

CrazyUtil 类只提供了一个添加天数的方法。本例的模板内容如下：

```
<span th:text="${#crazy.addDay('2018-02-24', 10)}"></span>
```

运行该模板后，输出结果如下：

```
<span>2018-03-02 10:54:42</span>
```

9.4.4 自定义执行属性

我们可以将一些公用的属性放到执行属性方言类中，这些定义好的属性都可以在处理器中获取，包括预处理器、一般处理器与后处理器。代码清单 9-26 定义了一个执行属性方言类。

代码清单 9-26：codes\09\custom-tpl\src\main\java\org\crazyit\boot\c9\exe\MyExeAttrDialect.java

```
public class MyExeAttrDialect implements IExecutionAttributeDialect {

    public String getName() {
        return "myUtil";
    }

    public Map<String, Object> getExecutionAttributes() {
        Map<String, Object> attrs = new HashMap<String, Object>();
        attrs.put("admin", "Angus");
```


仅供非商业用途或交流学习使用

```
        return attrs;
    }
}
```

在该方言类中设置了 admin 属性，你可以在处理器中获取该属性，如通过以下代码获取属性：

```
Map<String,Object> attrs = context.getConfiguration().getExecutionAttributes();
for(String key : attrs.keySet()) {
    System.out.println(key + "---" + attrs.get(key));
}
```

只要拿到模板上下文即可。一般处理器的 process 方法、处理者（Handler）的 getContext 方法都可以拿到上下文实例，在此不再赘述。

9.5 本章小结

本章主要讲解了 Thymeleaf 模板处理的原理，其核心是策略与责任链这两个设计模式，理解了这两个设计模式后，再去了解一下模板处理的流程，基本上就可以理解几大处理器的工作原理了。我们可以使用这几大处理器做很多事情，例如可以在预处理器中对 HTML 进行清洗，可以在一般处理器中处理自定义标签或属性，也可以在后处理器中修改 Thymeleaf 的处理结果。如果你只想开发自定义的标签或属性，而不想太深入了解 Thymeleaf 原理的话，则可以直接查看本章的 9.4 节的内容。

仅供非商业用途或交流学习使用

CHAPTER

10

第10章 Spring Boot与Thymeleaf整合

本章要点

- 在 Spring Boot 中配置 Thymeleaf
- 表单处理
- 片断的使用
- 处理器的使用
- Thymeleaf 与 WebFlow



仅供非商业用途或交流学习使用

在前面章节中已经简单讲述过 Spring Boot 与 Thymeleaf 的整合。我们在学习完 Thymeleaf 大部分的内容后，这一章将会深入学习 Spring Boot 与 Thymeleaf 的整合，包括如何在 Spring Boot 中配置模板引擎，如何进行表单处理等内容。Thymeleaf 与 Spring MVC 高度整合后，可以在 Web 应用领域中完全代替 JSP，我们可以使用 Spring 提供的控制器跳转到 Thymeleaf 模板，可以在模板中使用 Spring 表达式，还可以在 Spring MVC 中使用 Thymeleaf 的自定义标签。

10.1 在 Spring Boot 中配置 Thymeleaf

前面我们在使用 Thymeleaf 时，都是直接使用 API 来创建、配置 TemplateEngine。从本章开始，我们要在 Spring Boot 环境中使用 Thymeleaf。本章的所有 Spring Boot 项目的 pom.xml 文件内容如下：

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.1.RELEASE</version>
    </parent>
    <!-- 为当前项目引入 Starter 模块 -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>
    </dependencies>
</project>
```

以上的 pom.xml 文件在前面章节中已经介绍过。一般情况下，只需要使用 Thymeleaf 的 starter 模块。

10.1.1 模板引擎配置

这里所说的模板引擎配置，主要是对模板引擎对象（TemplateEngine）与模板解析器（ITemplateResolver）进行配置。我们在前面章节中，直接通过调用 API 来配置这两个对象，例如以下的代码片断：

```
        TemplateEngine templateEngine = new TemplateEngine();
        ClassLoaderTemplateResolver resolover = new ClassLoaderTemplateResolver();
        // 配置前缀与后缀
        resolover.setPrefix("/templates/");
        resolover.setSuffix(".html");
        // 设置模板解析器
        templateEngine.setTemplateResolver(resolover);
```

在默认情况下，Spring Boot 提供的自动配置模块，已经帮我们配置好 Thymeleaf。如果项

仅供非商业用途或交流学习使用

目环境中存在 SpringTemplateEngine 类，就会启动 Thymeleaf 的自动配置。配置类被加载之后，会读取 application.properties（或 yml）中的“spring.thymeleaf.*”配置，使用这些配置来创建模板解析器、模板引擎等对象，具体的配置项请参考本书的 5.5.1 节中的内容。另外，关于自动配置，可参考本书第 4 章中的内容。

当然，我们也可以自定义模板引擎的对象。在 Spring Boot 中，一般都使用注解的方式来配置 bean，请见代码清单 10-1。

代码清单 10-1：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\config\MyConfig.java

```
@Configuration
public class MyConfig {

    /**
     * 创建模板解析器的 bean
     */
    @Bean
    public SpringResourceTemplateResolver templateResolver(ApplicationContext ctx){
        SpringResourceTemplateResolver templateResolver = new
         SpringResourceTemplateResolver();
        // 设置解析器的属性，甚至可以使用自定义的属性
        templateResolver.setApplicationContext(ctx);
        templateResolver.setPrefix("classpath: /templates/");
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode(TemplateMode.HTML);
        templateResolver.setCacheable(true);
        return templateResolver;
    }

    /**
     *  创建模板引擎 bean
     */
    @Bean
    public SpringTemplateEngine templateEngine(ITemplateResolver templateResolver){
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver);
        templateEngine.setEnableSpringELCompiler(true);
        return templateEngine;
    }
}
```

但是，无论是自动配置 TemplateEngine，还是我们手工创建 TemplateEngine，最终是如何调用 process 来处理模板的呢？将 Spring 与 Thymeleaf 一起使用时，我们根本不需要直接调用 process 方法，视图对象会帮我们实现。下一小节会讲解 Spring 的视图与视图解析器，本节我们先测试一下在代码清单 10-1 中创建的 TemplateEngine 是否可用。直接在控制器方法中调用 TemplateEngine 的方法，请见代码清单 10-2。

代码清单 10-2：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\config\ThyApp.java

```
@SpringBootApplication
@Controller
public class ThyApp {

    public static void main(String[] args) {
        SpringApplication.run(ThyApp.class, args);
    }
```


仅供非商业用途或交流学习使用

```
    @Autowired
    private TemplateEngine engine;

    @GetMapping("/test/{name}")
    @ResponseBody
    public String test(@PathVariable String name) {
        // 创建上下文对象
        Context ctx = new Context();
        ctx.setVariable("name", name);
        // 处理模板，模板为 templates/test1.html
        String result = engine.process("test1", ctx);
        return result;
    }
}
```

代码清单 10-2 是一个 Spring Boot 的启动类，也是一个控制器。在 test 方法中，使用了注入的 TemplateEngine 来处理“src/main/resources/templates”下的 test1.html 文件，该模板的全部内容如下：

```
<span th:text="${name}" style="color: red;">Prototype name</span>
```

运行代码清单 10-2，在浏览器中访问：http://localhost:8080/test/crazyit，则可以看到浏览器的输出，网页源代码如下：

```
<span style="color: red;">crazyit</span>
```

10.1.2 Spring MVC 视图解析器

在 Spring MVC 中，提供了 View 与 ViewResolver 两个接口，分别代表视图与视图解析器。视图对象的主要工作是将 Model 中的数据交给视图层框架（例如 FreeMarker、Velocity 等）去处理，而视图解析器，则会根据模板的名称来找到特定的视图对象。

注意：这里讲的是 Spring 视图解析器（ViewResolver），并不是 Thymeleaf 的模板解析器（ITemplateResolver）。

在 Spring MVC 中内置了许多视图解析器，Thymeleaf 也有自己的实现。为了让大家深入了解 Thymeleaf 与 Spring MVC 处理模板的过程，下面我们将模拟 Thymeleaf 解析器的实现，并使用它的 API，实现一个最简单的视图解析器。代码清单 10-3 是一个视图对象的实现。

代码清单 10-3：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\resolver\MyView.java

```
/**
 * 视图对象
 * @author 杨恩雄
 */
public class MyView implements View {

    // 模板引擎对象
    private TemplateEngine templateEngine;

    // 模板名称
    private String template;

    private String contentType = "text/html;charset=UTF-8";
```


仅供非商业用途或交流学习使用

```
    public void setTemplateEngine(TemplateEngine templateEngine) {
        this.templateEngine = templateEngine;
    }

    public void setTemplate(String template) {
        this.template = template;
    }

    @Override
    public String getContentType() {
        return this.contentType;
    }

    @Override
    public void render(Map<String, ?> model, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        // 将参数设置到 Context
        Context ctx = new Context();
        for(String key : model.keySet()) {
            Object obj = model.get(key);
            ctx.setVariable(key, obj);
        }
        response.setContentType(this.contentType);
        // 直接使用 TemplateEngine 来处理模板
        templateEngine.process(template, ctx, response.getWriter());
    }
}
```

我们来看视图对象中的 render 方法，它先将 Model（Map 对象）中保存的参数设置到一个 Context 对象中，这是 Thymeleaf 的 Context 对象，最后会调用 TemplateEngine 的 process 方法来处理模板。现在的问题是，在 Spring MVC 的 Controller 中，一般会直接返回模板名称（字符串，例如“test”），那么，如何根据这个字符串，找到对应的视图对象呢？这需要实现视图解析器，请见代码清单 10-4。

代码清单 10-4：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\resolver\MyViewResolver.java

```
/**
 * 自定义视图解析器
 * @author 杨恩雄
 */
public class MyViewResolver extends AbstractCachingViewResolver {

    // 模板引擎对象
    private TemplateEngine templateEngine;

    // 视图类
    private Class viewClass = MyView.class;

    public void setTemplateEngine(TemplateEngine templateEngine) {
        this.templateEngine = templateEngine;
    }

    /**
     * 根据名称加载视图对象
     */
```

```
    @Override
    protected View loadView(String viewName, Locale locale) throws Exception {
        // 获取 bean 工厂实例
        AutowireCapableBeanFactory beanFactory = getApplicationContext()
                .getAutowireCapableBeanFactory();
        boolean beanExists = beanFactory.containsBean(viewName);                    ①
        MyView view;
        // 判断 bean 实例是否存在
        if (beanExists) {
            // 存在直接获取
            view = (MyView) beanFactory.getBean(viewName);                          ②
        } else {
            // 不存在则创建一个 View 实例
            MyView viewInstance = (MyView) BeanUtils
                    .instantiateClass(viewClass);
            view = (MyView) beanFactory.initializeBean(viewInstance, viewName); ③
        }
        // 设置模板名称与模板引擎
        view.setTemplateEngine(this.templateEngine);
        view.setTemplate(viewName);
        return view;
    }
}
```

视图解析器的 loadView 方法会先根据视图的名称去 Spring 容器中寻找 bean（代码清单中的①处），如果 bean 存在，就直接使用该 bean 作为当前视图的对象（代码清单中的②），如果 bean 不存在，则创建一个新的 bean，该 bean 的类型为我们前面定义的 MyView，最后为 view 对象设置模板引擎的名称。

为了让这个视图解析器生效，需要配置容器，即增加一个 bean，请见代码清单 10-5。

代码清单 10-5：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\resolver\ViewConfig.java

```
@Configuration
public class ViewConfig {

    /**
     * 创建自定义的视图处理器
     */
    @Bean
    public MyViewResolver myViewResolver(TemplateEngine templateEngine) {
        MyViewResolver resolver = new MyViewResolver();
        resolver.setTemplateEngine(templateEngine);
        return resolver;
    }
}
```

在创建自定义视图解析器时，需要使用容器中的 TemplateEngine，在本例中，Thymeleaf 的模板引擎对象由 Spring 自动配置。接下来，编写控制器与启动类，请见代码清单 10-6。

代码清单 10-6：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\resolver\ResolverApp.java

```
@SpringBootApplication
@Controller
public class ResolverApp {

    public static void main(String[] args) {
```

```
        SpringApplicationBuilder builder = new SpringApplicationBuilder
(ResolverApp.class);
        // 关闭 Thymeleaf 视图解析
        builder.properties("spring.thymeleaf.enabled=false");
        builder.run(args);
    }

    @GetMapping("/test/{name}")
    public String test(@PathVariable String name, Model model) {
        model.addAttribute("message", "Hello, " + name);
        return "myView";
    }
}
```

由于在项目中添加了 Thymleaf 的依赖，并且要使用我们自己的视图解析器，因此在启动时，需要将 Thymeleaf 的模板解析功能禁用。在控制器中编写一个简单的 test 方法，该方法会将路径参数组装为新的字符串放到 Model 中，最后返回的模板名称为“myView”。视图解析器在处理时，会先根据“myView”查找 bean，没有找到则会创建一个名称为“myView”、类型为 MyView 的 bean。在我们前面自定义的视图对象 MyView 的 render 方法中，就可以得到 Model 中设置的变量、模板名称。最后调用 TemplateEngine 的 process 方法处理模板。本例中的 myView.html 模板，仅仅将 message 输出，模板内容如下：

```
<span style="color: red;" th:text="${message}">Welcome, admin</span>
```

运行代码清单 10-6，在浏览器中访问：http://localhost:8080/test/crazyit，则可以看到浏览器的输出，网页源代码为：

```
<span style="color: red;">Hello, crazyit</span>
```

Thymeleaf 的视图与视图解析器更为复杂，在其实现中还涉及转换类、片断的处理。我们在这里实现这个“山寨版”的视图解析器，目的是让读者更了解 Spring MVC 与 Thymeleaf 的整合，实际上还是调用 TemplateEngine 的 process 方法处理模板、输出结果。

在此，读者只需要知道视图、视图解析器的职责即可：视图解析器根据名称找到对应的视图对象，而在视图中，则会调用模板引擎的 API 来处理模板、输出结果。

10.1.3 Thymeleaf 视图解析器

Thymeleaf 提供了 View 与 ViewResolver 的实现类，分别是 ThymeleafView 与 ThymeleafViewResolver，它们的逻辑跟我们前面自定义的视图解析器大致相同，View 接口负责调用 TemplateEngine 的 process 方法处理模板文件，ViewResolver 用来寻找具体视图（View）的 bean。

我们可以对这两个实现类进行配置，例如可以为 ThymeleafViewResolver 配置解析器执行顺序，请见代码清单 10-7。

代码清单 10-7：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\thrsv\ThViewConfig.java

```
    @Bean
    public ThymeleafViewResolver viewResolver(TemplateEngine engine) {
        // 配置解析器，设置执行顺序
        ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
        viewResolver.setTemplateEngine((SpringTemplateEngine)engine);
        // 默认值为 Integer.MAX_VALUE
```

```
        viewResolver.setOrder(1);
        // 关闭缓存
        viewResolver.setCache(false);
        return viewResolver;
    }
```

只有实现了 Ordered 接口的 ViewResolver 才可以设定执行顺序，数字越大，越在后执行。

我们还可以配置视图的 bean，根据前文所述，视图解析器会根据模板名称到 Spring 容器中寻找 bean，找到就直接使用，换言之，具体的某个视图，也可以使用 bean 来配置，请见代码清单 10-8。

代码清单 10-8：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\thrsv\ThViewConfig.java

```
    @Bean
    @Scope("prototype")
    public ThymeleafView testView() {
        System.out.println("=== 创建 TestView ===");
        ThymeleafView  view = new ThymeleafView("testView");
        // 设置公用的静态变量
        Map<String, String> staticVars = new HashMap<String, String>();
        staticVars.put("msgPrefix", "Welcome");
        view.setStaticVariables(staticVars);
        return view;
    }
```

在代码清单 10-8 中，我们自己创建了一个 ThymeleafView，设置了名称为 msgPrefix 的变量。只要在控制器方法中返回“testView”，就总会设置这个变量。

代码清单中的视图 bean 为非单态模式，其每次请求 Spring 容器都会拿到一个新的实例。注意，如果开启了缓存，就算使用@Scope(“prototype”)，也不会去拿这个 bean，它会先去缓存中查找，换言之，想测试这里的@Scope(“prototype”)，需要将缓存关闭。

编写控制器，返回“testView”视图，请见代码清单 10-9。

代码清单 10-9：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\thrsv\ThResolverApp.java

```
    @GetMapping("/test/{name}")
    public String test(@PathVariable String name, Model model) {
        model.addAttribute("name", name);
        return "testView";
    }
```

在模板中将“msgPrefix”（定义在 testView 的 bean 里面）和“name”两个变量输出，模板内容如下：

```
    <span th:text="${msgPrefix} + ', ' + ${name}" style="color: red;">Prototype name</span>
```

运行启动类 ThResolverApp（与控制器为同一个类），成功启动后在浏览器中访问：http://localhost:8080/test/crazyit，页面源码如下：

```
    <span style="color: red;">Welcome, crazyit</span>
```

注意：如果自定义了 ThymeleafView 的 bean，则需要再配置一个 ThymeleafViewResolver 的 bean，并且要设置它的执行顺序，否则将会抛出异常，异常信息为：Property 'locale' is required。

10.1.4 数据转换配置

在本书的 7.2 节，已经讲解了 Thymeleaf 的数据转换功能，在此，大致讲下转换的配置，更详细的内容，可以查阅本书的 7.2 节。代码清单 10-9 中配置了一个格式化器。

代码清单 10-9：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\conver\BigDecimalFormatter.java

```
public class BigDecimalFormatter implements Formatter<BigDecimal> {

    public String print(BigDecimal object, Locale locale) {
        return "$" + object.toString() + "$";
    }

    public BigDecimal parse(String text, Locale locale) throws ParseException {
        return null;
    }
}
```

被设置到 Model 中的 BigDecimla 类型参数，都会被 BigDecimalFormatter 进行格式化，本例中的模板内容如下：

```
<span th:text="${money}" style="color: red;">Prototype name</span></br>
<span th:text="${{money}}" style="color: red;">Prototype name</span></br>
<span th:text="*{{money}}" style="color: red;">Prototype name</span></br>
<span th:text="${#conversions.convert(money, 'String')}" style="color: red;">
Prototype name</span>
```

在模板中会将参数名称为“money”的变量输出四次，第一次只是简单地使用变量选择器，第二次则使用了“${{...}}”语法，第三次使用“*{{...}}”语法，第四次则使用#conversions 对象，实现控制器方法，请见代码清单 10-10。

代码清单 10-10：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\conver\ConverApp.java

```
@GetMapping("/conver")
public String test(Model model) {
    // 设置参数让模板显示
    model.addAttribute("money",
            new BigDecimal(100.99).setScale(2, BigDecimal.ROUND_HALF_UP));
    return "conver";
}
```

运行启动类，在浏览器中输入：http://localhost:8080/conver，浏览器输出如下：

```
100.99
$100.99$
$100.99$
$100.99$
```

根据结果可知，后面 3 次输出 money 变量的语法，效果相同，这里推荐使用“${{...}}”语法。建议在项目中使用统一的代码风格。

10.1.5 模板国际化

在实现模板国际化时，可以使用文本表达式(#{...})来输出 properties 文件里面的属性值。在一般环境下，我们返回的模板名称为“crazyit”，Thymeleaf 将会到 crazyit.html 所在的目录，寻找 crazyit.properties 文件，再根据模板中的 key 来找到具体的 value，请见以下名称为“crazyit.html”的模板：

```
<span th:text="#{jdbc.user}">This is prototype text.</span>
```

整合 Spring Boot 后，以上的 properties 文件读取规则将不会生效。在默认情况下，国际化的资源文件名称为“messages.properties”，换言之，在所有模板中使用的属性 key，都是从 messages.properties 文件中读取的。我们可以通过修改 application.properties 来改变资源名称，例如以下的配置：

```
spring.messages.basename=i18n/messages
```

在该配置下 Spring 会读取 src/main/resources/i18n 目录下的 messages.properties 作为资源文件，使用 spring.messages.basename 属性可配置多个属性文件，文件之间用逗号隔开即可。

在实践中，有可能需要动态设置 spring.messages.basename，例如需要读取某个目录下面所有的 properties 文件，此时我们可以使用自定义的 MessageSource 来实现，请见代码清单 10-11。

代码清单 10-11：codes\10\boot-config\src\main\java\org\crazyit\boot\c10\prop\PropConfig.java

```
@Configuration
public class PropConfig {

    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource mySource = new ResourceBundleMessageSource();
        // 读取 src/main/resources/i18n 目录下面全部的 properties
        try {
            // 获取 i18n 目录的路径
            URL path = PropConfig.class.getResource("/i18n");
            File file = new File(path.getPath());
            // 过滤 i18n 目录下面的文件
            File[] propFiles = file.listFiles(new FileFilter() {
                public boolean accept(File folderFile) {
                    // 只处理 properties 文件
                    if(folderFile.getName().indexOf(".properties") != -1) {
                        return true;
                    }
                    return false;
                }
            });
            String[] names = new String[propFiles.length];
            // 获取文件名，格式为“i18n/fileName”
            for(int i = 0; i < propFiles.length; i++) {
                String fileName = propFiles[i].getName();
                String fileNameWithoutSuffix = fileName.substring(0,
                  fileName.lastIndexOf("."));
                names[i] = "i18n/" + fileNameWithoutSuffix;
            }
            // 将文件名设置为 basename
            mySource.addBasenames(names);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return mySource;
    }
}
```

在代码清单 10-11 中自定义了一个 ResourceBundleMessageSource 类型的 bean，在创建该

bean 时，Thymeleaf 会读取类路径下“i18n”目录中的全部 properties 文件，形成字符串数组，再将其设置到 ResourceBundleMessageSource 对象中，这样，就可以实现动态加载 properties 文件。使用代码清单 10-11 的配置方式，要注意以下两点：

- spring.messages 的相关配置将不会生效。
- MessageSource 类型的 bean 的名称（方法名）必须为“messageSource”，否则将不会生效。例如将代码清单 10-11 中的 messageSource 方法改为 myMessageSource，虽然会创建 bean，但 Spring 容器将找不到名称为 messageSource 的 bean，因此国际化配置不会生效。

以上是静态或动态设置 properties 文件的方式。在测试国际化时，我们需要设置不同的 Locale，例如以下代码，在启动 Spring Boot 前设置默认的 Locale：

```
public static void main(String[] args) {
    Locale.setDefault(Locale.ENGLISH);
    SpringApplication.run(LocaleApp.class, args);
}
```

如果没有修改 spring.messages.basename（默认为 messages），则以上代码会去读取 messages_en.properties 文件。

10.2 表单处理

本节讲解在 Spring Boot 中 Thymeleaf 模板对表单的处理。

10.2.1 表单提交

为控制器添加两个表单相关的方法，一个是打开表单模板，另一个用于处理表单提交，请见代码清单 10-12。

代码清单 10-12：codes\10\form\src\main\java\org\crazyit\boot\c10\config\SubmitApp.java

```
@PostMapping("/submit")
public String submit(@ModelAttribute User user) {
    System.out.println("表单对象: " + user.getUserName());
    return "submit";
}

@GetMapping("/test/{form}")
public String openForm(@PathVariable String form, @ModelAttribute User user) {
    return form;
}
```

其中的 openForm 方法用于打开表单，submit 方法用于处理表单的提交，模板请见代码清单 10-13。

代码清单 10-13：codes\10\form\src\main\resources\templates\submit.html

```
<form method="post" th:action="@{/submit}" th:object="${user}">
    <table>
        <tr>
            <td><input type="text" name="userName" th:value="*{userName}"/></td>
        </tr>
        <tr>
```

仅供非商业用途或交流学习使用


```
            <td><input type="submit" value="提交"/></td>
        </tr>
    </table>
</form>
```

在模板中使用了 th:action、th:object 及 th:value 三个属性，其中 th:action 的值是链接表达式。不论应用中配置了什么样的“server.context-path”，Thymeleaf 都会自动帮我们处理这个 form 的 action。th:object 属性声明选定 user 对象，在 form 的范围内，可以使用星号表达式来输出 user 的属性。将代码清单 10-13 中的表单提交后，submit 方法会处理该表单。submit 方法的参数是一个 User 实例，该实例表示一个表单对象。启动服务器类（SubmitApp.java），在浏览器中访问：http://localhost:8080/test/submit，则可看到本例的表单。输入字符并提交表单后，可以在控制台与页面中看到效果。

注意：本例的模板会输出 user.userName 属性，但打开模板前并不存在 User 对象，因此在 openForm 方法的参数中要添加一个 User 对象，且需要使用@ModelAttribute 来修饰，否则，将会抛出异常，异常信息大致为：“org.springframework.expression.spel.SpelEvaluationException: EL1007E: Property or field 'userName' cannot be found on null”。

10.2.2 使用 th:field 属性

Thymeleaf 与 Spring MVC 整合后，提供了一个 th:field 属性，使用这个属性，可以减少我们一部分工作量。下面，我们在不同的表单域中使用这个属性，控制器代码请见代码清单 10-14。

代码清单 10-14：codes\10\form\src\main\java\org\crazyit\boot\c10\field\FieldApp.java

```
@GetMapping("/field")
public String field(Model model) {
    // 设置用户对象
    model.addAttribute("user", new User("crazyit", "lady"));
    // 设置性别常量，集合中存放字符串
    List<String> sexCons = new ArrayList<String>();
    sexCons.add("man");
    sexCons.add("lady");
    model.addAttribute("sexCons", sexCons);
    return "field";
}
```

在 Model 中设置了两个属性：一个是 User 对象，该对象被设置了名称“crazyit”和性别“lady”；另外一个是用于存放性别常量的集合，该集合对象主要用于下拉框显示，两个常量值分别为“man”与“lady”。模板内容请见代码清单 10-15。

代码清单 10-15：codes\10\form\src\main\resources\templates\field.html

```
<form method="post" th:action="@{/submit}" th:object="${user}">
    <table>
        <tr>
            <td><input type="text" th:field="*{userName}"/></td>
        </tr>
        <tr>
            <td><textarea th:field="*{userName}"></textarea></td>
        </tr>
        <select th:field="*{sex}">
            <option th:each="sexCon : ${sexCons}" th:value="${sexCon}" th:text=
              "${sexCon}"></option>
```

仅供非商业用途或交流学习使用


```
            </select>
            <tr>
                <td><input type="submit" value="提交"/></td>
            </tr>
        </table>
    </form>
```

在模板中有三个地方使用了 th:field 属性，分别是 input、textarea 与 select 这三个表单域。在下拉框中会对 option 进行遍历，这里使用性别的常量集合。启动服务器（FieldApp.java），在浏览器中访问：http://localhost:8080/field，网页源代码如下：

```
    <form method="post" action="/submit">
        <table>
            <tr>
                <td><input type="text" id="userName" name="userName" value="crazyit"/>
</td>
            </tr>
            <tr>
                <td><textarea id="userName" name="userName">crazyit</textarea></td>
            </tr>
            <select id="sex" name="sex">
                <option value="man">man</option>
                <option value="lady" selected="selected">lady</option>
            </select>
            <tr>
                <td><input type="submit" value="提交"/></td>
            </tr>
        </table>
    </form>
```

使用 th:field 属性后，三个节点都会生成 id 与 name 属性。其中在 input 中使用 th:field 属性，会再生成 value 属性；在 textarea 中使用 th:field 属性，会自动设置文本；在 select 中使用 th:field 属性，则会根据 User 的 sex 属性，自动选中相应的 option。使用 th:field 属性时，如果 HTML 节点中已经存在相应的属性，则不会再另外生成。

由此可见，th:field 属性就是一个工具属性，可以帮我们完成部分 HTML 代码的编写工作。

注意： th:field 属性需要使用星号表达式（*{...}），也就是说，要先使用 th:object 属性声明表单对象，再使用 th:field=*{...}”对表单域进行处理。

10.2.3 HTML 节点的 id 生成

我们已经知道，th:field 属性会自动帮我们生成节点的 id、name 属性，生成的 id 将是表单对象的某个属性名，那么对于 HTML 中的多选框（checkbox）、单选框（radio），id 属性是如何生成的呢？下面我们来学习 ids 对象的使用，看看 th:field 属性的 id 生成规则。

Thymeleaf 的内置对象#ids 主要用于 HTML 节点的 id 生成。在控制器中设置一个简单的属性值，请见代码清单 10-16。

代码清单 10-16：codes\10\form\src\main\java\org\crazyit\boot\c10\ids\IdsApp.java

```
    @GetMapping("/ids")
    public String field(Model model) {
        FormObject formObject = new FormObject();
        formObject.setName("crazyit");
        model.addAttribute("formObject", formObject);
```

仅供非商业用途或交流学习使用


```
        return "ids";
    }
```

在控制器方法中，设置一个 FormObject 对象到 Model 中，这只是一个普通的 JavaBean，模板请见代码清单 10-17。

代码清单 10-17：codes\10\form\src\main\resources\templates\ids.html

```
<form method="post" th:action="@{/submit}" th:object="${formObject}">
   <table>
      <tr>
         <td>
            <input type="text" th:field="*{name}" th:id="${#ids.seq('name')}"/>
            <input type="text" th:field="*{name}" th:id="${#ids.seq('name')}"/>
            <input type="text" th:field="*{name}" th:id="${#ids.seq('name')}"/>
         </td>
      </tr>
   <table>
</form>
```

处理模板后，生成的三个 text 结果如下：

```
            <input type="text" value="crazyit" id="name1" name="name"/>
            <input type="text" value="crazyit" id="name2" name="name"/>
            <input type="text" value="crazyit" id="name3" name="name"/>
```

根据结果可知，ids 对象帮我们生成的 id 属性，以“nameN”格式存在，索引 N 从 1 开始。ids 对象帮我们使用计算器来生成一个全模板唯一的 id。这种 id 策略很少应用在文本框中，更多地应用在 checkbox 或者 radio 中。修改代码清单 10-17，在后面追加输出 checkbox 与 radio，请见代码清单 10-18。

代码清单 10-18：codes\10\form\src\main\resources\templates\ids.html

```
<form method="post" th:action="@{/submit}" th:object="${formObject}">
   <table>
      <tr>
         <td>
            <input type="text" th:field="*{name}" th:id="${#ids.seq('name')}"/>
            <input type="text" th:field="*{name}" th:id="${#ids.seq('name')}"/>
            <input type="text" th:field="*{name}" th:id="${#ids.seq('name')}"/>
         </td>
      </tr>
      <tr>
         <td>
            <input type="checkbox" th:field="*{name}" value="*{name}"/>
            <input type="checkbox" th:field="*{name}" value="*{name}"/>
            <input type="checkbox" th:field="*{name}" value="*{name}"/>
         </td>
      </tr>
      <tr>
         <td>
            <input type="radio" th:field="*{name}" value="*{name}"/>
            <input type="radio" th:field="*{name}" value="*{name}"/>
            <input type="radio" th:field="*{name}" value="*{name}"/>
         </td>
      </tr>
   </table>
</form>
```

仅供非商业用途或交流学习使用

在后面追加的 checkbox 与 radio 里面，并没有使用 ids 对象，对于这两个节点，th:field 属性会自动使用计数器来生成 id。执行模板后，生成的 9 个 input 结果如下：

```
    <input type="text" id="name1" name="name" value="crazyit"/>
    <input type="text" id="name2" name="name" value="crazyit"/>
    <input type="text" id="name3" name="name" value="crazyit"/>

    <input type="checkbox" value="*{name}" id="name4" name="name"/><input type=
"hidden" name="_name" value="on"/>
    <input type="checkbox" value="*{name}" id="name5" name="name"/><input type=
"hidden" name="_name" value="on"/>
    <input type="checkbox" value="*{name}" id="name6" name="name"/><input type=
"hidden" name="_name" value="on"/>

    <input type="radio" value="*{name}" id="name7" name="name"/>
    <input type="radio" value="*{name}" id="name8" name="name"/>
    <input type="radio" value="*{name}" id="name9" name="name"/>
```

在我们的例子中，全部都使用了 name 属性，负责生成 id 的计数器从 1 开始，因此对 9 个 input 生成的 id 为“name1”到“name9”。当使用 checkbox 时，会自动在 checkbox 后面生成一个隐藏域，该隐藏域是自动生成的，目的是为了避免浏览器不发送没有选中的 checkbox 的值，从而造成服务获取的值为 null。举个例子，如果在 checkbox 后面没有这些隐藏域，使全部的 checkbox 都没有被选中，那么在后台获取到的数组为 null，如果加上这些隐藏域，在后台拿到的数组就不为 null，只是长度为 0 而已。

因此，我们需要知道，在 checkbox 与 radio 中使用 th:field 属性时，对于同一个表单对象的属性，会自动使用计数器来生成 id。

10.2.4 实现 label 节点

对于 checkbox 与 radio，还要解决另外一个问题，就是<label>节点的问题。使用 label 节点时，需要提供 checkbox 或者 radio 的 id，由于 id 是自动生成的，所以我们并不能知道代码中的 id 值。为了解决该问题，可以使用 ids 对象的 next 与 prev 方法。以代码清单 10-18 中的最后三个 radio 为例，为它们添加 lable 节点，请见代码清单 10-19。

代码清单 10-19：codes\10\form\src\main\resources\templates\ids.html

```
    <input type="radio" th:field="*{name}" value="*{name}"/>
    <label th:for="${#ids.prev('name')}" th:text="A 选项"/>
    <input type="radio" th:field="*{name}" value="*{name}"/>
    <label th:for="${#ids.prev('name')}" th:text="B 选项"/>
    <input type="radio" th:field="*{name}" value="*{name}"/>
    <label th:for="${#ids.prev('name')}" th:text="C 选项"/>
```

处理模板后，输出如下：

```
<input type="radio" value="*{name}" id="name7" name="name"/>                ①
<label for="name7">A 选项</label>
<input type="radio" value="*{name}" id="name8" name="name"/>                ②
<label for="name8">B 选项</label>
<input type="radio" value="*{name}" id="name9" name="name"/>                ③
<label for="name9">C 选项</label>
```

在以上结果中，计算器在①中的索引为 7，那么“A 选项”的 label 调用了 prev 方法，它

仅供非商业用途或交流学习使用

要获取前一个索引，也就是会拿到 7，同样地，B 选项与 C 选项的 label，分别得到 8 与 9，也就是说，prev 方法会获取前一个索引的值（input 在 label 前面）。

使用 next 方法与之类似，如果将代码清单 10-19 中的 prev 方法全部换成 next 方法，那么 A 选项将会得到 8（7 的后一个索引是 8），B 选项得到 9，C 选项得到 10。

我们只需要知道一个规则：不管是 prev 还是 next 方法，都会获取索引，而不会修改索引。

10.2.5 在模板中处理 checkbox 与 radio

前面讲过，th:field 会自动帮 checkbox 与 radio 生成唯一的 id。现在我们可以使用数据测试一下这两种表单域，根据实际的数据来实现选中效果。

举个最简单的例子，一个用户可以选择自己喜欢的颜色，也可以选择性别，根据该需求设计一个 ColorUser 对象，并设置两个属性，请见以下代码：

```
public class ColorUser {

    // 喜欢的颜色
    private String[] myColors;

    // 性别
    private String sex;

    ...省略 setter 与 getter 方法
}
```

编写控制器方法，在打开模板前，设置相应的数据，请见代码清单 10-20。

代码清单 10-20：codes\10\form\src\main\java\org\crazyit\boot\c10\form\FormApp.java

```
@GetMapping("/form")
public String field(Model model) {
    // 定义可选 的颜色
    String[] colors = new String[]{"red", "yellow", "green"};
    // 定义可选的性别
    String[] sexes = new String[]{"man", "lady"};
    // 将常量数据设置到 Model 中
    model.addAttribute("colors", colors);
    model.addAttribute("sexes", sexes);
    // 创建 User
    ColorUser user = new ColorUser();
    // 设置用户喜欢的颜色
    user.setMyColors(new String[]{"yellow", "green"});
    // 设置性别
    user.setSex("lady");
    // 将用户对象设置到 Model 中
    model.addAttribute("user", user);
    return "form";
}
```

在代码清单 10-20 的粗体字代码中使用的是常量，在实际应用中可能会从数据库表中读取这些值。后面创建了一个 ColorUser 的实例，并设置了喜欢的颜色与性别，常量数据与 ColorUser

仅供非商业用途或交流学习使用

都会被设置到 Model 中，模板内容请见代码清单 10-21。

代码清单 10-21：codes\10\form\src\main\resources\templates\form.html

```
<form method="post" th:action="@{/submit}" th:object="${user}">
    <table>
        <tr th:each="sexCon : ${sexes}">                                    ①
            <td>
                <input type="radio" th:field="*{sex}" th:value="${sexCon}"/>  ②
                <label th:for="${#ids.prev('sex')}" th:text="${sexCon}"></label>
            </td>
        </tr>
        <tr th:each="color : ${colors}">                                    ③
            <td>
                <input type="checkbox" th:field="*{myColors}" th:value="${color}"/>④
                <label th:for="${#ids.prev('myColors')}" th:text="${color}"></label>
            </td>
        </tr>
    </table>
</form>
```

①处的代码迭代性别常量，②处的代码使用 th:field 属性来设置 radio，th:field 属性会自动生成 id 与 name 属性。③处的代码与①处的类似，即迭代颜色常量，它使用的是 checkbox。启动服务器（FormApp.java），在浏览器中访问：http://localhost:8080/form，界面如图 10-1 所示。

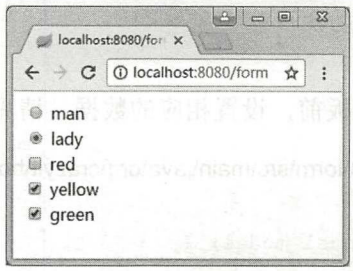


图 10-1 checkbox 与 radio

由图 10-1 可知，th:field 属性会自动帮我们将相应的 radio 与 checkbox 选中。查看源文件，部分内容如下：

```
<td>
    <input type="radio" value="man" id="sex1" name="sex"/>
    <label for="sex1">man</label>
</td>
<td>
    <input type="radio" value="lady" id="sex2" name="sex" checked="checked"/>
    <label for="sex2">lady</label>
</td>
<td>
    <input type="checkbox" value="red" id="myColors1" name="myColors"/><input
type="hidden" name="_myColors" value="on"/>
    <label for="myColors1">red</label>
</td>
<td>
    <input type="checkbox" value="yellow" id="myColors2" name="myColors"
checked="checked"/><input type="hidden" name="_myColors" value="on"/>
    <label for="myColors2">yellow</label>
</td>
```

仅供非商业用途或交流学习使用


```
    <td>
        <input type="checkbox" value="green" id="myColors3" name="myColors"
checked="checked"/><input type="hidden" name="_myColors" value="on"/>
        <label for="myColors3">green</label>
    </td>
```

对于该例，要注意以下细节：

- 在 checkbox 或 radio 节点中，th:field 属性传入的始终是表单对象（本例是 user）的特定属性。
- 注意 label 节点的 for 属性，如果 label 在 input 之后，则需要使用 ids 对象的 prev 方法获取前一个计算器的索引，如果 label 节点在 input 节点之前，则需要使用 next 方法。

10.2.6 表单提交处理 checkbox 与 radio

后台控制器处理表单提交比较简单，只要在上一节的模板基础上加上“提交”按钮即可。在控制器方法中使用@ModelAttribute 映射表单对象，就可以拿到 checkbox 与 radio 的值，控制器方法请见代码清单 10-22。

代码清单 10-22：codes\10\form\src\main\java\org\crazyit\boot\c10\form\FormApp.java

```
@PostMapping("/submit")
public String submit(@ModelAttribute ColorUser user, Model model) {
    // 设置常量到 Model 中
    setConstant(model);
    model.addAttribute("user", user);
    System.out.println("性别：" + user.getSex());
    System.out.println("喜欢颜色：" + Arrays.toString(user.getMyColors()));
    return "form";
}

private void setConstant(Model model) {
    // 定义可选的颜色
    String[] colors = new String[]{"red", "yellow", "green"};
    // 定义可选的性别
    String[] sexes = new String[]{"man", "lady"};
    // 将常量数据设置到 Model 中
    model.addAttribute("colors", colors);
    model.addAttribute("sexes", sexes);
}
```

本例的模板文件使用代码清单 10-21。启动服务器（FormApp.java）后，在浏览器中访问：http://localhost:8080/form，选中 checkbox 或者 radio，即可看到控制台的输出。

本节讲解了表单域的使用，重点讲述了 th:field 属性，这个属性可以帮我们完成很多工作，特别是对 select、checkbox 与 radio 使用时，作用更加明显。

10.3 表单验证

表单验证分为前端验证与服务器端验证，本节介绍服务器端的验证。

10.3.1 表单验证

很多时候，我们都需要对表单对象进行验证，例如在表单中输入的名称不能为空。除了会

仅供非商业用途或交流学习使用

在前端做 JavaScript 验证外，为了进一步保障系统的安全，也会在服务器端对表单对象进行验证。Java 提供了 JSR 303 规范，该规范主要用于数据验证，而 Hibernate Validator 实现了 JSR 303 规范。下面我们使用 Hibernate Validator 来实现表单的服务器端验证。

如果我们在项目依赖中加入了 spring-boot-starter-thymeleaf，默认就会加入 Hibernate Validator 的依赖。下面简单使用 Hibernate Validator 验证 3 个表单域，表单对象请见代码清单 10-23。

代码清单 10-23：codes\10\validate\src\main\java\org\crazyit\boot\c10\simple\User.java

```
public class User {

    @NotBlank(message = "{user.name.notBlank}")
    private String name;

    @Email(message = "{user.email}")
    private String email;

    @Size(min = 6, message = "{user.passwd.size}")
    private String passwd;

    ...省略 setter 与 getter 方法
}
```

在 User 中使用了 NotBlank、Email 与 Size 注解，name 属性不能为空，email 属性需要是邮箱格式，passwd 属性至少为 6 位。这 3 个属性被验证失败后，默认会到类路径下的 ValidationMessages.properties 文件中寻找对应的 key 作为错误信息。编写控制器方法，将错误信息输出，本例的控制器代码请见代码清单 10-24。

代码清单 10-24：codes\10\validate\src\main\java\org\crazyit\boot\c10\simple\ValidApp.java

```
    @PostMapping("/save")
    public String save(@Valid User user, BindingResult result) {
        // 控制台输出错误信息
        if(result.hasErrors()) {
            List<ObjectError> list = result.getAllErrors();
            for (ObjectError error : list) {
                System.out.println(error.getCode() + "---" + error.getDefaultMessage());
            }
        }
        return "validate";
    }

    @GetMapping("/open")
    public String openAdd(@ModelAttribute User user) {
        return "validate";
    }
```

在保存方法中，使用@Valid 来修饰表单对象 User，并且提供了 BindingResult 对象，该对象保存验证的结果。新建一个简单的模板，提供 3 个输入框架，本例模板内容请见代码清单 10-25。

代码清单 10-25：codes\10\validate\src\main\resources\templates\validate.html

```
<form method="post" th:action="@{/save}" th:object="${user}">
    name: <input type="text" th:field="*{name}"/></br>
```

仅供非商业用途或交流学习使用


```
    email: <input type="text" th:field="*{email}"/></br>
    passwd: <input type="text" th:field="*{passwd}"/></br>
    <input type="submit" value="提交"/>
</form>
```

启动服务器（ValidApp.java），在浏览器中访问：http://localhost:8080/open，如图 10-2 所示。

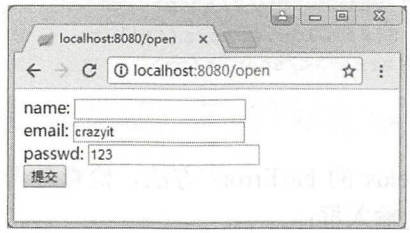


图 10-2 验证表单

提交表单后，控制台输出如下：

```
NotBlank---请输入用户名
Size---密码长度至少 6 位
Email---请输入 Email 格式
```

10.3.2 修改信息文件

如前所述，错误信息默认被配置在 ValidationMessages.properties 文件中。我们可以提供一个 bean，读取另外的属性文件，请见代码清单 10-26。

代码清单 10-26：codes\10\validate\src\main\java\org\crazyit\boot\c10\simple\MyConfig.java

```
@Configuration
public class MyConfig {

    @Autowired
    private  MessageSource messageSource;

    @Bean
    public LocalValidatorFactoryBean validator() {
        LocalValidatorFactoryBean validator = new LocalValidatorFactoryBean();
        // 使用系统默认的 messages.properties 文件
        validator.setValidationMessageSource(messageSource);
        return validator;
    }
}
```

代码清单中的 validator 方法新建了一个 LocalValidatorFactoryBean，并设置验证信息文件为系统的 messages.properties。

关于表单验证还有很多的内容，我们甚至可以自定义验证逻辑，但这并不是本书的重点，在此不再赘述。

10.3.3 输出错误信息

在前面的例子中，当验证失败时，可以通过 BindingResult 对象拿到相应的错误信息，并且我们可以在模板中使用#fields 对象将错误信息输出。注意，不是 th:field 属性，而是内置对

象#fields。修改模板文件，加入输出错误信息的 HTML，请见代码清单 10-27。

代码清单 10-27：codes\10\validate\src\main\resources\templates\validate.html

```
<form method="post" th:action="@{/save}" th:object="${user}">
    name: <input type="text" th:field="*{name}"/></br>
    email: <input type="text" th:field="*{email}"/></br>
    passwd: <input type="text" th:field="*{passwd}"/></br>
    <input type="submit" value="提交"/> <p></p>
    <div th:each="err : ${#fields.errors('*')}">
        <span th:text="${err}"></span></br>
    </div>
</form>
```

除此之外，还可以使用 fields 的 hasErrors 方法，检查是否有验证错误，若有，则修改表单域的 CSS 样式，请见以下的输入框：

```
<input type="text" th:field="*{name}" th:errorclass="fieldError"/>
<style>
    .fieldError {
        background-color: red;
    }
</style>
```

这里使用了 th:errorclass 属性，当存在错误时，其会将 input 的 class 设置为“fieldError”。前面的例子会在表单中输出错误信息，我们也可以在表单外输出错误信息或者判断信息是否存在，请见以下模板：

```
<form method="post" th:action="@{/save}" th:object="${user}">
    ...表单内容
</form>
<!-在表单外输出错误信息 -->
<div th:errors="${user.name}"></div>
<!-- 判断是否存在错误 -->
<div th:if="${#fields.hasErrors('${user.name}')}">name 属性存在错误</div>
```

我们可以把在模板中使用的 fields 对象当作 BindingResult 来使用，它除了可以判断是否存在错误并获取错误信息外，还可以得到具体的异常对象。调用 fields.detailedErrors 方法，会返回一个 DetailedError 对象的集合，在 DetailedError 对象中保存了错误信息、错误编码、字符等数据。

10.4　片断的使用

在前面章节中我们学习过片断的使用，这一节我们学习如何在 Spring Boot 中定义、引用 Thymeleaf 的片断。

10.4.1　引用片断

在片断文件中，可直接使用 th:fragment 属性来配置片断，再使用“~{模板::片断(参数)}”语法来引用片断，还可以使用选择器来引用不同的节点。在 Spring Boot 环境中使用片断与前面的方法类似，例如，我们在代码清单 10-28 中定义与引用一个片断。

代码清单 10-28：片断定义：codes\10\fragment\src\main\resources\templates\commonHead.html

```
<head th:fragment="head(title)">
    <link rel="stylesheet" type="text/css" th:href="@{/common.css}" />
    <title th:text="${title}"></title>
</head>
```

片断引用：codes\10\fragment\src\main\resources\templates\simple.html

```
<html>
    <head th:replace="~{commonHead :: head('测试标题')}"></head>
    <body>
        <span class="redText">使用 common.css 中的样式</span>
    </body>
</html>
```

在代码清单 10-28 中定义了一个带变量的片断，需要传入标题。在 simple.html 模板中，会直接使用 th:replace 替换掉整个 head 节点，而 commonHead.html 中的片断会引用 common.css。在 CSS 文件中定义了一个 redText 样式，将字体设置为红色。编写启动类与控制器，请见代码清单 10-29。

代码清单 10-29：codes\10\fragment\src\main\java\org\crazyit\boot\c10\simple\SimpleApp.java

```
@SpringBootApplication
@Controller
public class SimpleApp {

    public static void main(String[] args) {
        SpringApplication.run(SimpleApp.class, args);
    }

    @GetMapping("/simple")
    public String head() {
        return "simple";
    }
}
```

运行服务器，在浏览器中访问：http://locahost:8080/simple，浏览器输出如图 10-3 所示。



图 10-3 片断引用

除了以上的使用方式外，还可以使用片断选择器，根据 id、class 或其他条件选择引用的片断，详细请参考本书的 8.1 节中的内容。

10.4.2 使用 bean 选择片断

片断的配置较为灵活，我们可以将多个片断放到一个模板文件中，然后通过配置 Spring 的 bean 来选择特定片断，模板文件请见代码清单 10-30。

代码清单 10-30：codes\10\fragment\src\main\resources\templates\common.html

```
<head th:fragment="commonHead">
```

```
    <link rel="stylesheet" type="text/css" th:href="@{/common.css}" />
    <title>Fragment Title</title>
</head>
<footer th:fragment="commonFooter">
    <span>copy right</span>
</footer>
```

代码清单 10-30 在一个模板文件中定义了 head 与 footer 节点，它们都使用 th:fragment 配置了片断。如果我们只想使用 footer 节点，除了可以在其他模板中选中“commonFooter”片断外，还可以在代码中，将 footer 节点选中，并且将其作为一个 bean 放到容器中，请见代码清单 10-31。

代码清单 10-31：codes\10\fragment\src\main\java\org\crazyit\boot\c10\bean\MyConfig.java

```
@Configuration
public class MyConfig {

    @Bean
    public ThymeleafViewResolver viewResolver(TemplateEngine engine){
        // 配置解析器，设置执行顺序
        ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
        viewResolver.setTemplateEngine((SpringTemplateEngine)engine);
        // 关闭缓存
        viewResolver.setCache(false);
        viewResolver.setCharacterEncoding("UTF-8");
        return viewResolver;

    }

    @Bean(name="foot")
    @Scope("prototype")
    public ThymeleafView footView() {
        // 创建视图对象
        ThymeleafView view = new ThymeleafView("common");
        // 设置选中 commonFooter 片断
        view.setMarkupSelector("commonFooter");
        return view;
    }

    @Bean(name="head")
    @Scope("prototype")
    public ThymeleafView headView() {
        // 创建视图对象
        ThymeleafView view = new ThymeleafView("common");
        // 设置选中 commonHead 片断
        view.setMarkupSelector("commonHead");
        return view;
    }
}
```

在代码清单 10-31 中配置了两个 bean：foot 与 head，foot 这个 bean 将会从 common.html 模板中，选取“commonFooter”片断，head 这个 bean 会从 common.html 模板中，选取“commonHead”片断，两个 bean 都会返回 ThymeleafView，即视图对象。编写控制器代码，返回不同的视图，请见代码清单 10-32。

代码清单 10-32：codes\10\fragment\src\main\java\org\crazyit\boot\c10\bean\BeanApp.java

```
@GetMapping("/common/{name}")
public String head(@PathVariable String name) {
    // 根据路径参数来决定返回 head 还是 foot
    return name;
}
```

启动服务器（BeanApp.java），在浏览器中访问：http://localhost:8080/common/foot，或者访问：http://localhost:8080/common/head，通过查看网页源代码可以看到结果。

10.4.3 在控制器中引用片断

前面我们在模板中选取片断，通过 bean 来选取片断，实际上还可以通过控制器的方法返回值来选取片断，请见代码清单 10-33。

代码清单 10-33：codes\10\fragment\src\main\java\org\crazyit\boot\c10\ctl\CtlApp.java

```
@GetMapping("/common/simple")
public String simple() {
    return "common :: simpleBody";
}
```

控制器方法 simple 会返回“common::simpleBody”这样的字符串，这意味着 Thymeleaf 会到 common.html 模板中选取“simpleBody”片断。除此之外，还可以向片断传递参数，但传递参数的方式与模板引用不一样，请见代码清单 10-34。

代码清单 10-34：codes\10\fragment\src\main\java\org\crazyit\boot\c10\ctl\CtlApp.java

```
@GetMapping("/argbody/{name}")
public String argbody(@PathVariable String name) {
    return "common :: argbody (name = '" + name + "')";
}
```

传递参数时，需要使用“片断名称(argName = argValue)”这样的形式，多个参数之间使用逗号隔开。如果使用模板的传递参数的方式，则会抛出异常，例如以下的代码：

```
@GetMapping("/argbody/{name}")
public String argbody(@PathVariable String name) {
    return "common :: argbody ('test')";
}
```

以上的代码将会抛出异常，异常信息为：java.lang.IllegalArgumentException: Parameters in a view specification must be named (non-synthetic): 'common :: argbody ('test')'。

10.5 自定义标签

在 Spring Boot 环境中配置自定义标签，方法基本上与在一般环境中类似，都是建立方言类、处理类，再进行配置。这一节我们开发一个自定义标签。

10.5.1 方言与处理器

我们在 Thymeleaf 模板中，建立自定义标签，例如实现一个 JSTL 的 c:if 标签。处理类与方言类请见代码清单 10-35。

代码清单 10-35

标签处理器：codes\10\dialect\src\main\java\org\crazyit\boot\c10\dialect\MyTagProcessor.java

```
public class MyTagProcessor implements IElementTagProcessor {

    /**
     * 返回匹配的节点名称
     */
    public MatchingElementName getMatchingElementName() {
        ElementName eleName = ElementNames.forHTMLName("c:if");
        MatchingElementName mEleName = MatchingElementName.forElementName(
                getTemplateMode(), eleName);
        return mEleName;
    }

    /**
     * 根据属性匹配节点，含有 test 属性的节点都会被匹配到
     */
    public MatchingAttributeName getMatchingAttributeName() {
        AttributeName attName = AttributeNames.forHTMLName("", "test");
        MatchingAttributeName mAttName = MatchingAttributeName
                .forAttributeName(getTemplateMode(), attName);
        return mAttName;
    }

    /**
     * 返回模板模式
     */
    public TemplateMode getTemplateMode() {
        return TemplateMode.HTML;
    }

    /**
     * 返回处理器优先级
     */
    public int getPrecedence() {
        return 0;
    }

    /**
     * 处理<c:if test>的节点
     */
    public void process(ITemplateContext context, IProcessableElementTag tag,
            IElementTagStructureHandler structureHandler) {
        // 创建表达式解析器
        IStandardExpressionParser expressionParser = StandardExpressions
                .getExpressionParser(context.getConfiguration());
        // 获取属性值
        String expressionValue = tag.getAttributeValue("test");
        // 解析表达式
        final IStandardExpression expression = expressionParser
                .parseExpression(context, expressionValue);
        // 执行表达式
        final Object value = expression.execute(context);
        // 将结果转换为boolean 值
        boolean result = EvaluationUtils.evaluateAsBoolean(value);
        // 处理节点
```

```
            if (result) {
                // 若结果为 true，则仅仅删除标签
                structureHandler.removeTags();
            } else {
                // 若结果为 false，则删除整个节点
                structureHandler.removeElement();
            }
        }
    }
```

方言类：codes\10\dialect\src\main\java\org\crazyit\boot\c10\dialect\TagDialect.java

```
public class TagDialect implements IProcessorDialect {

    public String getName() {
        return "JSTL";
    }

    public String getPrefix() {
        return null;
    }

    public int getDialectProcessorPrecedence() {
        return 0;
    }

    public Set<IProcessor> getProcessors(String dialectPrefix) {
        Set<IProcessor> ps = new HashSet<IProcessor>();
        ps.add(new MyTagProcessor());
        return ps;
    }
}
```

该代码清单中的处理器会处理 c:if 标签，将标签表达式拿出来执行后，处理器根据执行结果，判断是否将整个节点删除。

10.5.2 配置方言

我们需要将实现的方言与处理器告诉 TemplateEngine。直接为 TemplateEngine 添加方言即可，请见代码清单 10-36。

代码清单 10-36：codes\10\dialect\src\main\java\org\crazyit\boot\c10\MyConfig.java

```
@Configuration
public class MyConfig {

    @Autowired
    TemplateEngine templateEngine;

    @PostConstruct
    public void init() {
        // 为模板引擎设置方言对象
        TagDialect myTabDialect = myTabDialect();
        templateEngine.addDialect(myTabDialect);
    }

    /**
     * 创建一个自定义方言 bean
     */
    @Bean
    public TagDialect myTabDialect() {
```

```
        return new TagDialect();
    }
}
```

为配置类注入 TemplateEngine，在 init 方法中，为模板引擎设置方言。编写控制器方法，设置测试参数，请见代码清单 10-37。

代码清单 10-37：codes\10\dialect\src\main\java\org\crazyit\boot\c10\DialectApp.java

```
@SpringBootApplication
@Controller
public class DialectApp {

    public static void main(String[] args) {
        SpringApplication.run(DialectApp.class, args);
    }

    @GetMapping("/index/{sex}")
    public String index(@PathVariable String sex, Model model) {
        model.addAttribute("sex", sex);
        return "index";
    }
}
```

模板文件内容如下：

```
<html>
    <body>
        <c:if test="${sex == 'man'}">
            <span>男</span>
        </c:if>
        <c:if test="${sex == 'lady'}">
            <span>女</span>
        </c:if>
    </body>
</html>
```

启动服务器，在浏览器中访问：http://localhost:8080/index/lady，可以看到输出结果。

在 Spring Boot 环境中配置自定义方言较为简单，与在一般环境中配置无异，只不过 Spring 已经帮我们创建、配置好了 TemplateEngine。

我们知道，模板会被一系列的 Handler（责任链）处理，在 Spring Boot 环境中也会有同样的处理，但启动责任链的方式与在一般环境中不同。在此我们不深入讲解处理原理，读者只需要知道，我们提供的 Handler 都会被放到责任链中执行，提供的一般处理器（Processor）都会被放到 ProcessTemplateHandler 中执行即可。

10.6 本章小结

这一章主要学习 Thymeleaf 与 Spring Boot 的整合使用，包括在 Spring Boot 环境中对 Thymeleaf 的配置、表单处理等知识。本章我们需要掌握如何配置模板引擎，如何处理各种表单域，特别是 th:field 属性的使用。学习完本章，你可以在 Spring Boot 环境中定制自己的模板引擎，可以使用 Spring Boot 与 Thymeleaf 实现大部分的 Web 功能。

CHAPTER

11

第 11 章
使用 JavaScript 库

本章要点

- 使用 jQuery
- 使用 Bootstrap
- 使用 Vue.js

在进行应用前端开发时，不可避免地要使用到一些 JavaScript 库，本章介绍几款较为流行的 JavaScript 库，你可以使用这些库（框架）来开发基本的页面功能，例如数据列表、表单处理、表单验证等。本章所介绍的每一个框架，都有一定的历史及知识体系，内容相当庞杂，但限于篇幅，本章仅对这些框架进行入门级的介绍，目标是让读者掌握最基本的功能开发，如果你已经掌握这些 JavaScript 库的使用，则可跳过本章。

11.1 使用 jQuery

这一节讲述 jQuery，这是近几年来最为流行的 JavaScript 库。

11.1.1 概述

jQuery 的第一个版本发布于 2006 年，它一直倡导使用简捷的方式编写 JavaScript 程序，就像其官网 Logo 所写的："write less, do more"。在 jQuery 的第一个版本发布以前，市面上充斥着许多优秀的 JavaScript 库，像 Dojo、ExtJS、Prototype 等。随着 AJAX 技术的兴起，这些 JavaScript 库被应用到各类的项目中，掌握一到两个这样的框架，已经成为对 Java Web 开发程序员的基本要求之一。jQuery 推出后，得到各大厂商的支持，其简洁、灵活的特性深受开发人员的追捧。近年来，jQuery 在众多的 JavaScript 库中脱颖而出，成为应用最广的 JavaScript 技术。在笔者成书时，jQuery 已经发布了 3.3.1 版本，本书所有使用 jQuery 的案例，均使用该版本。

本章的示例都是在 Spring Boot 与 Thymeleaf 的环境下运行的，JavaScript 文件都放在 src/main/resources/static/js 目录下，模板（页面）文件放在 src/amin/resources/templates 目录下。下面我们来编写一个最简单的模板文件，使用 jQuery 的 Hello World，请见代码清单 11-1。

代码清单 11-1：codes\11\jquery\src\main\resources\templates\simple.html

```
<html>
    <head>
        <script th:src="@{/js/jquery-3.3.1.min.js}"></script>
        <script>
            $(document).ready(function() {
                alert("hello");
            });
        </script>
    </head>
    <body></body>
</html>
```

代码清单 11-1 对应的 Controller 以及启动类较为简单，在此不贴出代码，对应的文件为 codes\11\jquery\src\main\java\org\crazyit\boot\c11\SimpleApp.java。运行该类，在浏览器中访问：http://localhost:8080/simple，随后可看到浏览器弹出的提示。

jQuery 的基本语法为：$(selector).action()，即通过选择器选中对象后，再执行相应的行为。下面我们来学习 jQuery 的选择器。

11.1.2 选择器

jQuery 的选择器基于 CSS 选择器。我们可以使用选择器根据不同的条件选取节点，例如根据节点 id、节点 class 或属性来选择。下面介绍几种较为常用的选择方式，请见代码清单 11-2。

代码清单 11-2：codes\11\jquery\src\main\resources\templates\select.html

```
<html>
    <head>
        <link rel="stylesheet" type="text/css" th:href="@{/css/common.css}" />
        <script th:src="@{/js/jquery-3.3.1.min.js}"></script>
    </head>
    <body>
        <!-- id 选择 -->
        <div id="idTest">Id Test</div>
        <input type="button" value="Id Test" onClick="alert($('#idTest').html())"/> ①
        <!-- class 选择 -->
        <div class="redText">Red Text 1</div>
        <div class="redText">Red Text 2</div>
        <input type="button" value="Class Test" onClick="alert($('.redText').length)"/> ②
        <!-- class 与节点 -->
        <input type="button" value="Class Div Test" onClick="alert($('div.redText').
         length)"/>                                                                    ③
        <!-- 获取子节点 -->
        <div id="parent">
            <span>Text 1</span>
            <span>Text 2</span>
        </div>
        <input type="button" value="Select Child" onClick="alert($('#parent span').
         length)"/>                                                                    ④
        <!-- 根据属性选取节点 -->
        <input type="button" value="Select Attr" onClick="alert($('div[class=
         redText]').length)"/>                                                         ⑤
    </body>
</html>
```

在该代码清单中使用了 5 种选择方式。代码①根据 id 选取节点并输出其 HTML；代码②根据 class 选取节点并输出结果长度，由于模板中存在多个“redText”的 div，因此结果为 2；代码③选取所有 class 为 redText 的 div 节点；代码④选取 id 为“parent”的节点下面的所有 span 节点；代码⑤选取所有 class 为 redText 的 div 节点，效果与代码③一致，不过它通过属性的方式来选取节点。运行 Simple.java 类，在浏览器中访问：http://localhost:8080/select，随后可通过单击相应按钮查看效果。

选择器的功能非常强大，它的选取方式有很多种，远不止该例子中所列的这几种，在此不再赘述，你在使用的过程中，可根据具体的需求去查阅选择器的相关说明文档。

11.1.3 事件

选取了 HTML 节点后，可以为其指定事件，常见的鼠标事件有：click、dblclick、mouseenter、mouseleave，键盘事件有：keypress、keydown、keyup，表单事件有：blur、focus 等。在代码清单 11-3 中指定了 click 事件。

代码清单 11-3：codes\11\jquery\src\main\resources\templates\event.html

```
<script>
    $(document).ready(function(){
        $("#span").click(function() {
            alert("点击了 span");
        });
    });
```

```
</script>
```

在该代码清单中，id 为“span”的节点被单击后，会弹出一个提示。绑定事件较为简单，这里不再赘述。我们在系统开发中使用较多的是表单事件，例如表单域获取焦点（focus）、失去焦点（blur）事件，以及提交表单时使用的 submit 事件，读者可以根据具体需求使用相应的事件。

11.1.4 数据列表

一般情况下，当需要对后台数据进行遍历时，可使用 th:each 属性来完成。但在很多情况下，我们的数据是通过 AJAX 获取的，很多时候得到的是 JSON 或 XML 格式的数据，这时我们就需要使用 JavaScript 来处理数据。在代码清单 11-4 中，编写了一个使用 jQuery 展示数据列表的功能。

代码清单 11-4：codes\11\jquery\src\main\resources\templates\list.html

```
<script th:src="@{/js/jquery-3.3.1.min.js}"></script>
<script>
    $(document).ready(function(){
        // 模拟数据，在实际开发中这些数据可能是通过 AJAX 获取的
        var datas = [{"id":"1", "name":"疯狂 Spring Cloud 微服务架构实战",
                    "author":"杨恩雄"},
                    {"id":"2", "name":"疯狂工作流讲义", "author":"杨恩雄"}];
        // 删除模板中的数据
        $("#dataTable").empty();
        // 遍历 JSON 数组
        for(var i = 0; i < datas.length; i++) {
            // 获取单个数据对象
            var data = datas[i];
            // 创建 tr 与 td
            var tr = $("<tr style='height: 40px;'></tr>");
            var idTd = $("<td>" + data.id + "</td>");
            var nameTd = $("<td>" + data.name + "</td>");
            var authorTd = $("<td>" + data.author + "</td>");
            // 依次为 tr 添加 3 个 td
            tr.append(idTd).append(nameTd).append(authorTd);
            // 添加到数据列表中
            $("#dataTable").append(tr);
        }
    });
</script>
<table style="width: 500px; text-align: center;" cellspacing="0" border="1">
    <thead>
        <tr>
            <td width="10%">id</td>
            <td width="60%">书名</td>
            <td width="30%">作者</td>
        </tr>
    </thead>
    <tbody id="dataTable">
        <!-- 静态数据 -->
        <tr style="height: 40px;">
            <td>1</td>
            <td>疯狂 Spring Cloud 微服务架构实战</td>
```

```
                <td>杨恩雄</td>
            </tr>
        </tbody>
    </table>
```

在该代码清单中定义了一个 JSON 数组(datas 变量),该数组会被添加到列表的 tbody 中。在遍历数据时，先创建一个 tr，再依次创建 3 个 td 并添加到 tr 里面，最后将 tr 添加到 tbody 中，在这里每一条数据都会变成列表中的一行。有一个小细节，在添加数据前，会先将原来列表中的静态数据删除，这样做减小了对原型界面的破坏。运行启动类 SimpleApp，在浏览器中访问：http://localhost:8080/list，结果如图 11-1 所示。

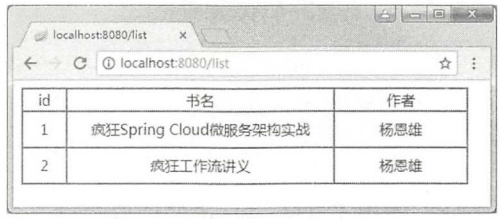


图 11-1 数据列表

11.1.5 AJAX 调用 GET 方法

jQuery 提供了多个 AJAX 方法，我们可以直接请求服务获取数据，比较常用的是调用 HTTP 的 GET 与 POST 方法。下面我们先使用 jQuery 调用 GET 方法。编写获取数据的服务，请见代码清单 11-5。

代码清单 11-5：codes\11\jquery\src\main\java\org\crazyit\boot\c11\SimpleApp.java

```
@GetMapping("/ajax")
public String ajax() {
    return "ajax";
}

@GetMapping(value = "/books", produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public List<Book> getBooks() {
    List<Book> books = new ArrayList<Book>();
    Book b1 = new Book("1", "疯狂工作流讲义", "杨恩雄");
    Book b2 = new Book("2", "疯狂 Spring Cloud 微服务架构实战", "杨恩雄");
    books.add(b1);
    books.add(b2);
    return books;
}
```

代码清单中的 getBooks 方法发布了一个返回 JSON 的服务，Spring 会将该方法返回的对象转换为 JSON 字符串。启动服务器，在浏览器中访问：http://localhost:8080/books，输出结果如下：

```
[{"id":"1","name":"疯狂工作流讲义","author":"杨恩雄"},{"id":"2","name":"疯狂 Spring Cloud 微服务架构实战","author":"杨恩雄"}]
```

服务已经可以正常访问。编写模板文件，使用 jQuery 发送 AJAX 请求调用该服务，模板内容请见代码清单 11-6。

代码清单 11-6：codes\11\jquery\src\main\resources\templates\ajax.html

```
<script th:src="@{/js/jquery-3.3.1.min.js}"></script>
<script>
    // 页面初始化时请求数据
    $(document).ready(function(){
        // 使用 Thymeleaf 内联语法，设置请求路径
        var url = "[[@{/books}]]";
        // 调用 GET 方法获取数据
        $.get(url, function(datas, status){
            for(var i = 0; i < datas.length; i++) {
                var data = datas[i];
                // 简单输出数据，可将数据写入 table 等列表中
                alert(data.id + "-" + data.name + "-" + data.author);
            }
        });
    });
</script>
```

在模板中构建请求 URL 时，使用了 Thymeleaf 的内联语法，在该语法中使用链接表达式配置了请求路径，最后调用 jQuery 的 GET 方法获取数据。启动服务器，在浏览器中访问：http://localhost:8080/ajax，可以看到浏览器的提示信息。

11.1.6 AJAX 调用 POST 方法

我们先使用$.ajax 方法，向服务器端发送 POST 请求，服务器端代码请见代码清单 11-7。

代码清单 11-7：codes\11\jquery\src\main\java\org\crazyit\boot\c11\SimpleApp.java

```
@PostMapping(value = "/save", consumes = MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public Book save(@RequestBody Book book) throws Exception {
    System.out.println("接收的数据：" + book.getId() + "-" + book.getName() + "-"
            + book.getAuthor());
    return book;
}
```

服务器方法仅仅将 Book 对象打印到控制台，再返回该对象。在页面中，使用 jQuery 发送请求，请见代码清单 11-8。

代码清单 11-8：codes\11\jquery\src\main\resources\templates\post.html

```
<script th:src="@{/js/jquery-3.3.1.min.js}"></script>
<script>
    function send() {
        var url = "[[@{/save}]]";
        var data = {id: "3", name: "Thymeleaf 实战", author: "杨恩雄"};
        $.ajax({
            type: "POST",
            dataType: "json",
            contentType: "application/json; charset=utf-8",
            url: url,
            data: JSON.stringify(data),
            success: function(data) {
                alert(data.name);
            }
        });
    }
```

```
</script>
<input type="button" value="Send Post" onClick="send()"/>
```

与前面的 GET 方法类似，使用 Thymeleaf 内联语法构建 URL，再使用$.ajax 方法发送请求，整个过程比较简单。在实际开发时，我们所发送的请求数据，有可能是从表单中获取的，数据类型可能会更加复杂，这里只需要掌握如何发送请求即可。

除了$.ajax 方法外，还可以使用$.post 方法，但如果向服务器发送 JSON 数据，还需要额外设置 contentType，否则服务器将无法接收数据，以下的代码效果与代码清单 11-7 一致：

```
<script>
    $.ajaxSetup({
        contentType: "application/json; charset=utf-8"
    });
    function send() {
        var url = "[[@{/save}]]";
        var data = {id: "3", name: "Thymeleaf 实战", author: "杨恩雄"};
        $.post(url, JSON.stringify(data), function(data) {
            alert(data.name);
        });
    }
</script>
```

至于使用$.ajax 还是$.post，可自行选择，本例使用$.post 方法。启动服务器（SimpleApp.java），在浏览器中访问：http://localhost:8080/post，单击按钮即可查看效果。

11.1.7 表单验证

jQuery 强大的地方在于其插件，我们可以将一些通用的功能提取出来形成插件，目前已经有许多现成的插件供我们选择，在此给大家介绍一个较为常用的插件：表单验证（jQuery Validation）。本书使用的验证插件版本为 1.17.0，读者可以到本书的共享 soft 目录中下载该版本的插件，文件名称为 jquery-validation-1.17.0.zip，也可以直接到项目中获取相应的 JavaScript 文件。代码清单 11-9 中的模板文件即使用了验证插件。

代码清单 11-9：codes\11\jquery\src\main\resources\templates\validate.html

```
<script th:src="@{/js/jquery-3.3.1.min.js}"></script>
<script th:src="@{/js/jquery.validate.min.js}"></script>
<script th:src="@{/js/messages.js}"></script>
<form id="userForm">
    <table>
        <tr>
            <td>用户名：</td>
            <td><input type="text" name="name"/></td>
        </tr>
        <tr>
            <td>密码：</td>
            <td><input type="text" name="passwd"/></td>
        </tr>
        <tr>
            <td>邮箱：</td>
            <td><input type="text" name="email"/></td>
        </tr>
        <tr>
            <td> </td>
```

```
                <td><input type="submit" value="提交"/></td>
            </tr>
        </table>
    </form>
    <script>
        $("#userForm").validate({
            rules: {
                name: {
                    required: true
                },
                passwd: {
                    minlength: 6
                },
                email: {
                    email: true
                }
            },
            messages: {
                name: {
                    required: "要写名字"
                }
            }
        });
    </script>
```

代码清单 11-9 中有一个表单，该表单中有 3 个表单域：名称、密码和邮箱。使用 validate 方法对其进行验证，验证规则为：name 必填，passwd 至少为 6 个字符，email 需要为合法的邮箱格式。其中在模板中定义了 name 表单域的必填提示信息，而 passwd 与 email 的提示信息，则使用在 messages.js 中定义的提示信息。messages.js 提供信息的国际化文件，其大致内容如下：

```
(function( factory ) {
    if ( typeof define === "function" && define.amd ) {
        define( ["jquery", "../jquery.validate"], factory );
    } else if (typeof module === "object" && module.exports) {
        module.exports = factory( require( "jquery" ) );
    } else {
        factory( jQuery );
    }
}(function( $ ) {

$.extend( $.validator.messages, {
    required: "这是必填字段",
    email: "请输入有效的电子邮件地址",
    minlength: $.validator.format( "<span style='color: red;'>最少要输入 {0} 个字符
</span>" ),
    ... 省略其他提示信息
} );
return $;
}));
```

我们下载 jQuery Validation 插件的压缩包后，可以在压缩包的 dist/localization 目录下找到其他的国际化文件，使用时将其引入到页面即可。

启动服务器（SimpleApp.java），在浏览器中访问：http://localhost:8080/validate，填写表单

测试具体的验证效果。这里有一个小细节需要注意，minlength 的验证提示，被设置了红色字体的 CSS 样式，读者在使用时，可以根据具体需求设置不同的样式。

11.2 Bootstrap

这一节学习使用 Bootstrap。

11.2.1 概述

Bootstrap 是一个开源的前端框架，由 Twitter 的设计师开发，Bootstrap 提供了一整套界面组件。在 Bootstrap 的框架内，我们只需要使用少量的代码，就可以使用、配置这些界面组件。Bootstrap 兼容 jQuery，自带了多个 jQuery 插件，且大部分的插件都可以直接使用。使用 Bootstrap，我们可以轻松构建出拥有良好交互性的系统界面，而不需要编写大量的 JavaScript 与 CSS 代码。Bootstrap 还提供了响应式布局，我们使用 Bootstrap 所编写的界面，可以兼容多种设备。

在笔者成书时，Bootstrap 已经发布了 4.0.0 版本，本书的示例都使用该版本。我们先使用 Bootstrap 编写一个简单的界面，模板文件如代码清单 11-10 所示。

代码清单 11-10：codes\11\bootstrap\src\main\resources\templates\simple.html

```
<!doctype html>
<html lang="en">
   <head>
      <link rel="stylesheet" type="text/css" th:href="@{/css/bootstrap.min.css}" />
      <script th:src="@{/js/jquery-3.3.1.min.js}"></script>
      <script th:src="@{/js/bootstrap.min.js}"></script>
   </head>
   <body>
      <!-- 列表 -->
      <table class="table">
         <thead>
            <tr>
               <td>id</td>
               <td>书名</td>
               <td>作者</td>
            </tr>
         </thead>
         <tbody>
            <tr>
               <td>1</td>
               <td>疯狂 Spring Cloud 微服务架构实战</td>
               <td>杨恩雄</td>
            </tr>
            <tr>
               <td>2</td>
               <td>疯狂工作流讲义</td>
               <td>杨恩雄</td>
            </tr>
         </tbody>
      </table>
      <!-- 分页 -->
      <ul class="pagination">
         <li class="page-item"><a class="page-link" href="#">Previous</a></li>
         <li class="page-item"><a class="page-link" href="#">1</a></li>
```

```
            <li class="page-item"><a class="page-link" href="#">2</a></li>
            <li class="page-item"><a class="page-link" href="#">3</a></li>
            <li class="page-item"><a class="page-link" href="#">Next</a></li>
        </ul>
        <!-- 按钮 -->
        <button type="button" class="btn btn-primary">测试按钮</button>
    </body>
</html>
```

代码清单 11-10 所示的页面实际上并不简单，其中有一个列表、一个分页组件及一个按钮，这些组件都使用了 Bootstrap 的样式。这里的服务器启动文件为 codes\11\bootstrap\src\main\java\org\crazyit\boot\c11\BootApp.java。启动服务器，在浏览器中访问：http://localhost:8080/simple，界面如图 11-2 所示。



图 11-2　使用 Bootstrap 组件

11.2.2　数据列表

下面，我们编写一个例子，从后台获取数据集合，在前台页面通过 Thymeleaf 的标签及 Bootstrap 的列表显示数据。首先在控制器中设置数据集合，请见代码清单 11-11。

代码清单 11-11：codes\11\bootstrap\src\main\java\org\crazyit\boot\c11\BootApp.java

```
@GetMapping("/list")
public String list(Model model) {
    List<Book> books = new ArrayList<Book>();
    books.add(new Book("1", "疯狂 Spring Cloud", "杨恩雄"));
    books.add(new Book("2", "疯狂工作流讲义", "杨恩雄"));
    model.addAttribute("books", books);
    return "list";
}
```

在模板中使用 th:each 遍历数据并输出，请见代码清单 11-12。

代码清单 11-12：codes\11\bootstrap\src\main\resources\templates\list.html

```
<head>
    <link rel="stylesheet" type="text/css" th:href="@{/css/bootstrap.min.css}" />
    <script th:src="@{/js/jquery-3.3.1.min.js}"></script>
    <script th:src="@{/js/bootstrap.min.js}"></script>
</head>
<body>
    <!-- 列表 -->
    <table class="table">
```

```
            <thead>
                <tr>
                    <td>id</td>
                    <td>书名</td>
                    <td>作者</td>
                </tr>
            </thead>
            <tbody>
                <tr th:each="book : ${books}" th:object="${book}">
                    <td th:text="*{id}">1</td>
                    <td th:text="*{name}">疯狂 Spring Cloud 微服务架构实战</td>
                    <td th:text="*{author}">杨恩雄</td>
                </tr>
            </tbody>
        </table>
    </body>
```

在代码清单 11-12 所示的模板中，使用 th:each 遍历集合，使用 th:object 指定对象，并且在每一列输出字段时，使用星号表达式。关于 Thymeleaf 的属性及表达式，已经在前面章节中详细讲解过，这里不再赘述。

11.2.3 分页

前面实现了一个简单的数据列表，我们再进一步，以数据列表为基础，实现简单的分页功能。首先设计一个数据对象。本例的数据都被封装在一个对象中，请见代码清单 11-13。

代码清单 11-13：codes\11\bootstrap\src\main\java\org\crazyit\boot\c11\DataPackage.java

```
public class DataPackage {

    // 列表数据
    private List datas;

    // 当前页
    private int currentPage;

    // 每页的数据量大小
    private int pageSize = 5;

    // 总数据量
    private int dataCount;

    // 总页数，由总数据量与每页的数据量计算得出
    private int pageCount;

    /**
     * 计算总页数
     */
    public void setPageCount() {
        if(dataCount % pageSize == 0) {
            this.pageCount = dataCount / pageSize;
        } else {
            this.pageCount = dataCount / pageSize + 1;
        }
    }
```

```
    ...省略 setter 与 getter 方法
}
```

在代码清单 11-13 中设计了一个 DataPackage 对象，用于存放数据集以及分页的相关属性（例如当前页、每页显示的数据量、总数据量等）。这里注意总页数（pageCount），该属性由总数据量（dataCount）和每页显示的数据量（pageCount）计算得出，我们直接在 DataPackage 里面提供 setPageCount 方法来计算总页数。接下来，编写控制器方法与模板查询方法，请见代码清单 11-4。

代码清单 11-14：codes\11\bootstrap\src\main\java\org\crazyit\boot\c11\BootApp.java

```
@GetMapping("/page/{currentPage}")
public String page(@PathVariable int currentPage, Model model) {
    DataPackage datas = findBooks(currentPage);
    model.addAttribute("datas", datas);
    return "page";
}

/**
 * 根据当前页查找数据（在实际应用中到数据库查找）
 */
private DataPackage findBooks(int currentPage) {
    DataPackage dp = new DataPackage();
    // 设置数据总数（在实际中可到数据库中查询，例如使用 select count）
    dp.setDataCount(30);
    dp.setCurrentPage(currentPage);
    // 模拟数据库查询数据
    List<Book> books = new ArrayList<Book>();
    for (int i = ((currentPage - 1) * dp.getPageSize()); i < (currentPage * dp
            .getPageSize()); i++) {
        Book book = new Book(String.valueOf(i), "book-" + i, "杨恩雄");
        books.add(book);
    }
    dp.setDatas(books);
    // 根据数据量等值，计算总页数
    dp.setPageCount();
    return dp;
}
```

代码清单 11-14 中的 findBooks 方法，会根据当前页参数查询数据。本例不涉及数据库，因此会模拟数据查询，读者不用太关注模拟的逻辑，只要知道调用该方法后的结果即可，例如，如果当前是第 1 页，则返回 id 为 0~4 的 Book 数据，如果是第 2 页，则返回 id 为 5~9 的 Book 数据。在模拟时，将数据总量设置为 30 条。接下来编写模板，请见代码清单 11-15。

代码清单 11-15：codes\11\bootstrap\src\main\resources\templates\page.html

```
<!doctype html>
<html lang="en">
    <head>
        <link rel="stylesheet" type="text/css" th:href="@{/css/bootstrap.min.css}" />
        <script th:src="@{/js/jquery-3.3.1.min.js}"></script>
        <script th:src="@{/js/bootstrap.min.js}"></script>
    </head>
    <body>
        <!-- 列表 -->
        <table class="table">
```

```
            <thead>
                <tr>
                    <td>id</td>
                    <td>书名</td>
                    <td>作者</td>
                </tr>
            </thead>`
            <tbody>
                <tr th:each="book : ${datas.datas}" th:object="${book}">
                    <td th:text="*{id}">1</td>
                    <td th:text="*{name}">疯狂 Spring Cloud 微服务架构实战</td>
                    <td th:text="*{author}">杨恩雄</td>
                </tr>
            </tbody>
        </table>
        <ul class="pagination">
            <li class="page-item active"
                th:class="${datas.currentPage == page} ? 'page-item active' : 'page-item'" ①
                th:each="page : ${#numbers.sequence(1, datas.pageCount)}">          ②
                <a class="page-link" href="#" th:href="@{/page/} + ${page}" th:text=
                  "${page}">1</a>                                                   ③
            </li>
        </ul>
        <div>
            <span th:text="|当前页：| + ${datas.currentPage}"></span></br>
            <span th:text="|总页数：| + ${datas.pageCount}"></span></br>
            <span th:text="|总数量：| + ${datas.dataCount}"></span>
        </div>
    </body>
</html>
```

原来的列表界面不需要进行大改，只需要把遍历的对象改为 DataPackage 即可。我们来看分页的处理，在一个 ul 节点中设置其 class="pagination"。①处的代码使用了三目运算符，如果是当前页，则 li 节点的 class 为"page-item active"，否则 class 为"page-item"。②处的代码使用#numbers 对象，从 1 到总页数对 li 节点进行循环，例如总页数为 6，则会输出 6 个 li 节点。③处的代码设置了 th:href 与 th:text 属性，它们会将循环的页数值输出。

运行代码清单 11-14（BootApp.java），在浏览器中访问：http://localhost:8080/page/1，界面如图 11-3 所示。



图 11-3 分页效果

分页功能还会涉及其他的一些周边功能，例如添加“前一页”与“后一页”按钮。另外，如果页数过多，还要根据特定的情况显示可单击的页码。对于这些功能，我们不在此实现，你可以根据具体的需求，编写 JavaScript 或者使用 Thymeleaf 的属性来实现。

11.2.4 表单

下面，我们使用 Bootstrap 实现一个简单的表单提交，表单提交的方式与前面的例子无异，不过表单相关的样式会有所不同，代码清单 11-16 是其控制器方法。

代码清单 11-16：codes\11\bootstrap\src\main\java\org\crazyit\boot\c11\BootApp.java

```
@GetMapping("/form")
public String form(@ModelAttribute Book book) {
    return "form";
}

@PostMapping(value = "/save")
public String save(@ModelAttribute Book book) {
    System.out.println("book: " + book.getName() + ", " + book.getAuthor()
            + ", " + book.getPrice());
    return "form";
}
```

模板代码请见代码清单 11-17。

代码清单 11-17：codes\11\bootstrap\src\main\resources\templates\form.html

```
<form name="bookForm" th:action="@{/save}" method="post" th:object="${book}">
    <table>
    <tr>
    <td>书名：</td>
        <td><input type="text" class="form-control" name="name" th:value="*{name}"/></td>
    </tr>
    <tr>
        <td>作者：</td>
        <td><input type="text" class="form-control" name="author" th:value="*{author}"/></td>
    </tr>
    <tr>
        <td>售价：</td>
        <td class="input-group">
        <div class="input-group-prepend">
            <span class="input-group-text">¥</span>
        </div>
        <input type="text" class="form-control" name="price" th:value="*{price}">
        <div class="input-group-append">
            <span class="input-group-text">元</span>
        </div>
        </td>
    </tr>
    <tr>
        <td> </td>
        <td><input type="submit" class="btn btn-outline-primary" value="提交"/></td>
    </tr>
    </table>
</form>
```

代码清单 11-17 是一个表单，表单中有 3 个文本框与一个“提交”按钮，它们都使用了

Bootstrap 的样式，其中要注意“售价”的文本框，在其前面和后面分别加入了一个 div，用于显示“¥”符号与“元”字，这种结构在 Bootstrap 中被称为“Input group”。运行代码清单 11-15，在浏览器中访问：http://localhost:8080/form，结果如图 11-4 所示。



图 11-4 Bootstrap 表单

输入各个字段的值后，单击“提交”按钮，则可以在控制台中看到输出。

11.2.5 警告框的使用

Bootstrap 提供了很多插件，我们在此结合前面的表单，使用 Bootstrap 的警告框插件设计一个警告框，控制器方法请见代码清单 11-18。

代码清单 11-18：codes\11\bootstrap\src\main\java\org\crazyit\boot\c11\BootApp.java

```
@GetMapping("/ajax")
public String ajax(@ModelAttribute Book book) {
    return "ajax";
}

@PostMapping(value = "/ajaxSave")
public String ajaxSave(@ModelAttribute Book book, Model model) {
    if("abc".equals(book.getAuthor())) {
        model.addAttribute("msg", "请输入正确的作者名称");
    }
    return "ajax";
}
```

控制器方法 ajaxSave 的逻辑很简单，当书本对象的作者名称为“abc”时，就设置 msg 变量的值，该变量的值将会在页面中显示。模板文件内容请见代码清单 11-19。

代码清单 11-19：codes\11\bootstrap\src\main\resources\templates\ajax.html

```
<form name="bookForm" th:action="@{/ajaxSave}" method="post" th:object="${book}">
    <div id="alertDiv" class="alert alert-warning alert-dismissible fade" role=
      "alert">
      <strong>错误：</strong> <span th:text="${msg}"></span>
    <a href="#" class="close" id="closeButton">x</a>
    </div>
    <table style="width: 100%;">
    <tr>
        <td>书名：</td>
        <td><input type="text" class="form-control" name="name" th:value=
         "*{name}"/></td>
    </tr>
    <tr>
```

```
            <td>作者：</td>
            <td><input type="text" class="form-control" name="author" th:value=
             "*{author}"/></td>
        </tr>
        <tr>
            <td>售价：</td>
            <td class="input-group">
            <div class="input-group-prepend">
                <span class="input-group-text">¥</span>
            </div>
            <input type="text" class="form-control" name="price" th:value="*{price}">
            <div class="input-group-append">
                <span class="input-group-text">元</span>
            </div>
            </td>
        </tr>
        <tr>
            <td> </td>
            <td><input type="submit" class="btn btn-outline-primary" value="提交"/> </td>
        </tr>
        </table>
    </form>
    <script>
    $(function() {
        var msg = '[[${msg}]]';
        if(msg == "") {
            // 无提示信息，隐藏提示 div
            $("#alertDiv").addClass("hide");
        } else {
            // 有提示信息，显示 div
            $("#alertDiv").addClass("show");
        }

        // 错误提示框关闭按钮的单击事件
        $('#closeButton').click(function() {
            $(".alert").addClass("hide");
            $(".alert").removeClass("show");
        });
    });
    </script>
```

先看下代码清单 11-19 的粗体字部分，这里定义了一个 div，这是一个用于显示错误信息的 div，这个 div 的 class 使用了 Alert 插件提供的样式。然后来看代码清单 11-18 最下面的 JavaScript 脚本，该脚本在页面初始化时，通过 Thymeleaf 的内联语法获取 msg 的值，如果有值，则为信息 div 的 class 添加“show”，如果没有值，则添加“hide”，我们从字面上就可以知道，有信息时，就会显示这个 div。

运行代码清单 11-18（BootApp.java），在浏览器中访问：http://localhost:8080/ajax，在“作者”的文本框中输入“abc”，则可以看到浏览器输出如图 11-5 所示。

以上通过提交表单的方式来使用警告框，我们还可以通过 AJAX 的方式来使用警告框，只需要直接使用 jQuery 相关的 AJAX 方法提交表单，并在回调时显示（或者隐藏）提示信息 div 即可。要显示提示框，为这个 div 添加“show”class 属性值，隐藏提示框，为这个 div 添加“hide”class 属性值。



图 11-5 显示警告框

Bootstrap 主要用于界面布局，其提供了很多的组件，如果你对 jQuery 比较熟悉，则使用 Bootstrap 也不是难事。在 Bootstrap 中，你可以使用很简洁的代码开发出拥有良好交互性的界面。

11.3 Vue.js

11.3.1 概述

Vue.js 是目前比较热门的一个 JavaScript 库。我们知道，jQuery 库可以直接操作 HMTL 节点实现具体的功能，而 Vue.js 则不一样，它会通过特别的 HTML、JavaScript 语法，来绑定 HTML 节点与具体的数据，当数据发生改变时，HTML 内容也会相应地更新。在使用 Vue.js 时，你会感觉它并不像一个 JavaScript 库，甚至有点像我们在前面学习的 Thymeleaf。本书使用的 Vue.js 版本为 2.5.17。下面我们开发一个 Vue 程序，请见代码清单 11-20。

代码清单 11-20：codes\11\vue\src\main\resources\templates\simple.html

```
<div id="app">
    <span>{{ message }}</span>
    <span v-text="person"></span>
</div>
<script th:src="@{/js/vue.min.js}"></script>
<script>
    var app = new Vue({
        el: "#app",
        data: {
            message: "Hello ",
            person: "crazyit"
        }
    });
</script>
```

HTML 中有一个 id 为“app”的 div，然后在 JavaScript 脚本里面，创建一个 Vue 实例，使用 el 来绑定 div。el 是这个 Vue 实例所维护的 DOM 元素，可以为其传入 CSS 选择器，它本身也可以是一个具体的 HTMLElement 实例，例如，el: document.getElementById("app")。Vue 对象维护一个 data 实例，用于存放数据，本例中我们声明了 message 与 person，然后在 div 里面，分别使用“{{...}}”与“v-text”来输出这两个数据值，其中 v-text 是 Vue.js 的指令之一。在代码清单中，通过 JavaScript 绑定节点并定义数据，在 HTML 中显示数据，当数据发生改变

时，HTML 显示的内容也会同步更新。为代码清单 11-20 加入一个按钮，并且添加操作方法，请见以下代码：

```
<span v-text="person"></span>
<input type="button" value="change" onclick="change()"/>
<script>
    var app = new Vue({
        el: "#app",
        data: {
            message: "Hello ",
            person: "crazyit"
        }
    });
    function change() {
        app.$data.person = "Spring";
    }
</script>
```

在 change 方法中，修改 Vue 实例中 data 的 person 数据，将其修改为“Spring”。由于 HTML 节点与数据已经绑定，因此我们并不需要操作 HTML 节点，就可以实现 HTML 节点内容的更改。

本例的主程序类为 codes\11\vue\src\main\java\org\crazyit\boot\c11\SimpleApp.java，运行后在浏览器中访问：http://localhost:8080/simple，则可以看到具体效果。使用 Vue.js 时，我们要改变以前使用 jQuery 等框架的习惯，不要直接操作 HTML 节点。

11.3.2 组件

Vue.js 的组件有点像 JSP 自定义标签，我们可以使用 Vue 对象的 component 方法来注册一个对象，然后直接在 HTML 中使用这个组件。下面编写一个简单的例子，页面内容请见代码清单 11-21。

代码清单 11-21：codes\11\vue\src\main\resources\templates\cpn.html

```
<div id="container">
    <crazyit></crazyit>
</div>
<script>
    Vue.component("crazyit", Vue.extend({
        template: "<span>{{ person }}: {{ age }}</span>",
        data: function() {
            return {
                person: "crazyit",
                age: 10
            }
        }
    }));
    new Vue({
        el: "#container"
    });
</script>
```

在代码清单 11-21 中，使用 Vue.component 注册了一个组件，该组件的模板为一个 span 节点，该节点会输出 person 与 age。在组件里面定义 data 时，会返回一个函数，这与在根实例中定义 data 不一样，需要注意。定义组件后，直接在 HTML 中，将组件作为 HTML 节点使用，

就像我们使用自定义的Thymeleaf或JSP标签一样。

除此之外，还可以为组件增加参数，请见以下代码片断：

```
<div id="container">
    <crazyit sex="男" height="180"></crazyit>
</div>
<script>
    Vue.component("crazyit", Vue.extend({
        props: ["sex", "height"],
        template: "<span>{{ person }}: {{ age }}, {{ sex }}, {{ height }}</span>",
        data: function() {
            return {
                person: "crazyit",
                age: 10
            }
        }
    }));
    new Vue({
        el: "#container"
    });
</script>
```

在以上代码中，在定义组件时使用了props来配置参数。在使用“crazyit”节点时，可以传入sex与height两个属性，这样在模板中就会使用“{{...}}”语法输出参数值。

无论是在根实例中定义data，还是在组件中定义data，都可以使用在Model中设置的变量，例如可以这样获取变量：

```
        data: function() {
            return {
                person: "crazyit",
                age: 10,
                school: '[[ ${school} ]]'
            }
        }
```

运行 codes\11\vue\src\main\java\org\crazyit\boot\c11\SimpleApp.java，在浏览器中访问：http://localhost:8080/cpn，则可以看到具体效果。

以上的例子在定义模板时，是直接设置template的值，如果模板的内容较多，则可以将其定义在script节点中，请见以下的代码：

```
<script type="text/x-template" id="my-template">
    <span>{{ person }}: {{ age }}, {{ sex }}, {{ height }}, {{ school }}</span>
</script>
<script>
    Vue.component("crazyit", Vue.extend({
        props: ["sex", "height"],
        template: "#my-template",
        data: function() {
            return {
                person: "crazyit",
                age: 10,
                school: '[[ ${school} ]]'
            }
        }
```


仅供非商业用途或交流学习使用

```
        }));
        new Vue({
            el: "#container"
        });
    </script>
```

以上的代码在 script 节点中定义了模板 id，调用 Vue.extend 时，设置一下 template 属性并直接传入模板 id 即可实现对模板的数据渲染。对于较为复杂的模板，可以考虑使用这种方式。

11.3.3 指令

在前面的第一个例子中，我们接触过 v-text 指令，其实 Vue.js 提供了许多指令，我们可以使用这些指令完成逻辑判断、循环、节点操作等工作，它们就像 Thymeleaf 模板中的节点属性一样。下面这些指令是它的常用指令。

- v-text：将值作为普通文本输出。
- v-html：将值作为 HTML 输出。
- v-show：如果属性值为 true，则使用该指令的节点将会显示。
- v-if：逻辑判断指令，与之类似的还有 v-else、v-else-if 指令。
- v-for：遍历指令，用于遍历并输出数据，类似于 th:each 属性。
- v-on：用于绑定事件监听器，例如 v-on:click 可缩写成@click。
- v-bind：用于绑定节点属性，可用“:”代替，例如 v-bind:src 可缩写成:src。
- v-model：使用在表单中，将表单域进行数据绑定。
- v-once：节点内容只会被处理一次，后面的数据变化将不会再次处理。

下面，我们编写代码来使用这几个指令，请见代码清单 11-22。

代码清单 11-22：codes\11\vue\src\main\resources\templates\direct.html

```
<div id="app">
    <span v-text="html"></span></br>
    <span v-html="html"></span></br>
    <!-- 根据 isLady 判断是否显示 span -->
    <span v-show="isLady">女</span></br>
    <!-- 判断 isLady 值，如果是 true 则显示该 span -->
    <span v-if="isLady == true">女</span></br>
    <!-- 为 button 绑定事件 -->
    <input type="button" value="确认" v-on:click="hello()"/></br>
    <!-- 如果 isLady 为 true，则 class 使用 redText -->
    <span v-bind:class="{ redText : isLady }">女</span></br>
    <!-- 数据改变后，将不会改变这个 span -->
    <span v-once v-html="html"></span></br>
</div>
<input type="button" value="改变" onclick="change()"/></br>
<script th:src="@{/js/vue.min.js}"></script>
<script>
    var app = new Vue({
        el: "#app",
        data: {
```


仅供非商业用途或交流学习使用

```
            html: "<span style='color: red;'>crazyit</span>",
            isLady: true
        }
    });
    function hello() {
        alert("hello");
    }
    function change() {
        app.$data.html = "<span style='color: blue;'>crazyit</span>";
    }
</script>
```

在代码清单 11-22 中的 div 中，使用了前面介绍的大部分指令。运行服务器（SimpleApp.java），在浏览器中访问：http://localhost:8080/direct，则可以看到页面效果。

11.3.4 循环指令

对于一些数据集合，可以使用 v-for 指令进行遍历，该指令的使用方法与 Thymeleaf 的 th:each 属性类似。我们编写一个简单的列表，用于遍历 JSON 数据，请见代码清单 11-23。

代码清单 11-23：codes\11\vue\src\main\resources\templates\each.html

```
<table id="app" style="width: 400px;">
    <thead>
        <tr>
            <td width="20%">1</td>
            <td width="60%">书名</td>
            <td width="20%">作者</td>
        </tr>
    </thead>
    <tbody>
        <tr v-for="book in books">
            <td>{{ book.id }}</td>
            <td>{{ book.name }}</td>
            <td>{{ book.author }}</td>
        </tr>
    </tbody>
</table>
<script th:src="@{/js/vue.min.js}"></script>
<script>
    // JSON 数据
    var datas = [{"id":"1", "name":"疯狂 Spring Cloud 微服务", "author":"杨恩雄"},
                 {"id":"2", "name":"疯狂工作流讲义", "author":"杨恩雄"}];
    var app = new Vue({
        el: "#app",
        data: {
            books: datas
        }
    });
</script>
```

在代码清单 11-23 中，在创建 Vue 实例时，将 datas 这个 JSON 数组放到 Vue 的 data 中，数据集被命名为 books。在 HTML 的 table 节点里面，使用 v-for 指令对 books 进行遍历，再使

仅供非商业用途或交流学习使用

用“{{...}}”语法输出各个属性。在这个例子中，可以使用 AJAX 获取 datas 数据，也可以从 SpringMVC 的 Model 中获取。运行 SimpleApp.java，在浏览器中访问：http://localhost:8080/each，则可以看到列表的效果。

11.3.5 表单验证

Vue.js 支持使用插件，我们在此仅介绍一个简单的表单验证插件，其名称为“vee-validate”，读者可以在 https://github.com/baianat/vee-validate 中找到这个项目。下面，编写页面，对表单进行验证，请见代码清单 11-24。

代码清单 11-24：codes\11\vue\src\main\resources\templates\form.html

```
<div id="app">
    <form>
        <table>
            <tr>
                <td>名称：</td>
                <td>
                    <input v-validate="'min: 3|max: 10|required'" name="name"
                     type="text">
                    <span v-show="errors.has('name')">{{ '最少 3 个字符，最长 10 个字
                     符' }}</span>
                </td>
            </tr>
            <tr>
                <td>作者：</td>
                <td>
                    <input v-validate="'required'" name="author" type="text">
                    <span v-show="errors.has('author')">{{ '请输入作者' }}</span>
                </td>
            </tr>
            <tr>
                <td> </td>
                <td><input type="submit" value="提交"/></td>
            </tr>
        </table>
    </form>
</div>
<script th:src="@{/js/vue.min.js}"></script>
<script th:src="@{/js/vee-validate.js}"></script>
<script>
    Vue.use(VeeValidate);
    new Vue({
        el: "#app"
    });
</script>
```

代码清单 11-24 中有两个表单域，其中“名称”字段的验证规则为：必填，长度需要在 3～10 个字符之间。“作者”字段是一个必填字段。在使用插件前，需要使用 Vue.use 方法进行安装。启动服务器，在浏览器中访问：http://localhost:8080/form，测试验证效果。除了这里介绍的 rquired、min 与 max 规则外，还有其他的规则，你可以查看“vee-validate”文档了解相关规则。

仅供非商业用途或交流学习使用

11.4 本章小结

本章主要介绍了 jQuery、Bootstrap 与 Vue.js 三个 JavaScript 库，它们都是目前较为常用的前端框架。我们可以使用这三个框架来开发页面功能，例如数据列表、表单、AJAX 等。在一个业务系统中，这些功能很常见。我们在本章所学的知识已经足以应付一般的功能开发，更深入的内容，我们会在后面章节学习。下一章，我们将学习数据库操作。

仅供非商业用途或交流学习使用

CHAPTER

12

第12章 数据库实战

本章要点

- 介绍 Spring Data
- Spring Data 与 JPA
- Spring Data 与 MongoDB
- Spring Data 与 Redis



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

在前面的章节中，我们都是围绕 MVC 框架、模板、JavaScript 等，讲述如何进行功能开发。本章我们讲解系统开发中最重要的部分：数据存储。无论是企业应用还是互联网应用，都离不开数据存储，本章将以 Spring Data 项目为基础，讲述如何在应用中进行数据库操作。

12.1 概述

12.1.1 关于 Spring Data

Spring Data 框架的目标是为数据访问提供一个通用的模型。无论数据库使用的是哪种技术，无论访问哪种数据库，在 Spring Data 中，都可以使用同样的方式，以同样的代码风格来实现对数据库的访问，这些数据库包括关系型数据库和非关系型数据库。

读者刚接触 Spring Data 时，会有种 Spring Data 要取代 Hibernate、Jedis 等数据库访问技术的感觉，实际上 Spring Data 以这些技术为基础，进行了进一步的封装，提供类似适配器的功能，使我们访问数据库变得更加简单，使代码风格更加统一。Spring Data 与数据库的关系如图 12-1 所示。

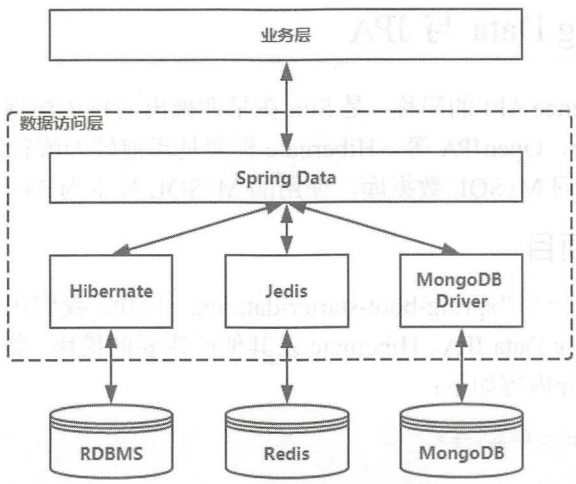


图 12-1 Spring Data 与数据库的关系

在传统的 Java EE 三层架构（控制层（Action）、业务层（Service）、数据访问层（DAO）中，Spring Data 为 DAO 层提供统一的数据访问模型。对于业务层的开发人员来说，大多数情况下不需要关心底层使用的是哪种数据访问技术。这样的设计，最大的好处是实现了各层的解耦，如果应用程序需要更换数据库，在理论上只需要很少的工作量就可以完成。

12.1.2 Spring Data 的功能

Spring Data 主要有以下几个功能：

- 提供数据与对象映射的抽象层，同一个对象，可以被映射为不同数据库的数据。
- 根据数据存储接口的方法名，自动实现数据查询。
- 为各个领域模型提供最基本的实现，例如普通的增、删、改、查功能。
- 可在原有逻辑的基础上，实现自定义的数据库操作逻辑。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

12.1.3 Spring Data 模块

使用 Spring Data 可以实现对不同数据库的访问，例如关系型数据库、MongoDB、Redis、Cassandra 等。Spring Data 提供了不同的模块来实现对不同数据库的访问。本章使用 MySQL、MongoDB、Redis 数据库，所涉及的模块如下。

- Spring Data JPA：该模块提供基本的数据操作功能，使用它可减少数据访问层的开发工作量。
- Spring Data MongoDB：使用该模块可轻松操作 MongoDB 数据库。
- Spring Data Redis：使用该模块可操作 Redis 数据库，并且操作过程得到了极大的简化。

除了以上 3 个模块外，Spring Data 官方还提供了很多其他的模块，在此不一一介绍。另外，除了官方发布的模块外，还有几个由社区发布的模块，由此可见，Spring Data 得到了开源社区的大力支持。

注意：本章中的大部分示例，实现的都是数据操作功能，并不涉及页面功能。

12.2 Spring Data 与 JPA

JPA 是 Java Persistence API 的简称，是 Sun 在早期推出的持久层规范，目前实现 JPA 规范的主流框架有 Hibernate、OpenJPA 等。Hibernate 框架是当前较为流行的 JPA 实现之一，在本节中我们将使用它来访问 MySQL 数据库，使用的 MySQL 版本为 5.6。

12.2.1 构建项目

Spring Boot 提供了一个“spring-boot-starter-data-jpa”模块，我们在项目中使用这个模块，就可以简单地整合 Spring Data JPA、Hibernate 及其他所需要的模块。新建名称为“spring-jpa”的项目，pom.xml 的部分内容如下：

```
<!-- Spring Boot 版本 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.1.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
        </dependency>
    </dependencies>
```

加入“spring-boot-starter-data-jpa”模块后，其会自动帮我们引入 Hibernate5、Spring Data JPA 等包。新建 application.yml 配置文件，配置内容如下：

```
server:
  port: 8080
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/SPRING_DATA
    username: root
    password: 123456
```

本例会为数据库连接一个名称为“CRA_PERSON”的表，映射的实体为 Person 类。CRA_PERSON 表主要有四个字段：ID（int）、NAME（varchar）、AGE（int）与 COMPANY（varchar），你可以将“codes\12\spring-jpa\src\main\resources\CRA_PERSON.sql”导入相应的数据库中。本例的项目结构如图 12-2 所示。

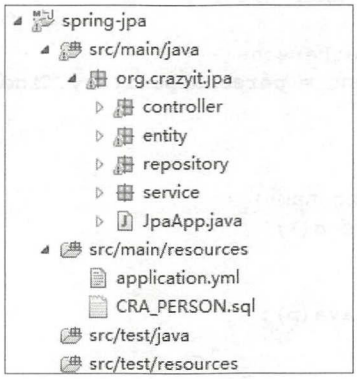


图 12-2 项目结构

其中，controller 下存放控制器，entity 下存放映射实体，service 下存放业务层对象，repository 下存放数据库访问对象。

12.2.2 数据访问层与业务层

映射“CRA_PERSON”表的 Person 类请见代码清单 12-1。

代码清单 12-1：codes\12\spring-jpa\src\main\java\org\crazyit\jpa\entity\Person.java

```
@Entity
@Table(name = "CRA_PERSON")
public class Person {
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    private Integer age;

    private String company;
    …省略 setter 和 getter 方法
}
```

Person 类是一个简单的 Java 对象，其使用 JPA 的相关注解进行修饰。下面在 repository 包中新建数据访问接口 PersonRepository，请见代码清单 12-2。

代码清单 12-2：codes\12\spring-jpa\src\main\java\org\crazyit\jpa\repository\PersonRepository.java

```
public interface PersonRepository extends JpaRepository<Person, Integer> {

}
```

PersonRepository 中并无任何接口方法和实现类。下面编写新增和查询的业务方法，请见代码清单 12-3。

代码清单 12-3：codes\12\spring-jpa\src\main\java\org\crazyit\jpa\service\PersonService.java

```
@Service
public class PersonService {

    @Autowired
    PersonRepository personRepository;

    public List<Person> getPersons() {
        List<Person> persons = personRepository.findAll();
        return persons;
    }

    public void save(String name) {
        Person p = new Person();
        p.setName(name);
        p.setAge(33);
        personRepository.save(p);
    }
}
```

在 PersonSevice 中注入 PersonRepository。PersonRepository 在继承了 JpaRepository 接口后，就拥有了最基本的数据库 CRUD 操作。初次接触它的朋友可能觉得很神奇，实际上 Spring 会为 PersonRepository 接口生成代理类，默认使用 JDK 动态代理。

默认使用 Spring MVC 控制器，可直接在控制器中注入 PersonService 的实例。运行“codes\12\spring-jpa\src\main\java\org\crazyit\jpa\JpaApp.java”，在浏览器中访问：http://localhost:8080/save/myName，向数据库中写入一条记录，再使用浏览器访问：http://localhost:8080/persons，查看具体的 JSON 数据。

注意：此处的数据查询、插入等功能，都是直接通过控制器来实现的，在本章的后面，会整合 Thymeleaf 来实现数据的增、删、改、查功能。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

12.2.3 自定义数据存储逻辑

Spring 帮我们生成的代理类，虽然可以完成很多工作，但在实际应用中，有时仍需要实现我们自己的数据存储逻辑。新建一个 PersonRepositoryCustom 接口，让原来的 PersonRepository 接口继承它，在 PersonRepositoryCustom 接口中加入需要自定义的接口方法，请见代码清单 12-4。

代码清单 12-4：codes\12\spring-jpa\src\main\java\org\crazyit\jpa\repository\PersonRepositoryCustom.java

```
public interface PersonRepositoryCustom {

    List<Person> myQuery();
}
```

修改原来的 PersonRepository 接口，让其同时继承 PersonRepository 和 JpaRepository。新建一个名称为“PersonRepositoryImpl”的实现类，实现 PersonRepositoryCustom 接口，如代码清单 12-5 所示。

代码清单 12-5：codes\12\spring-jpa\src\main\java\org\crazyit\jpa\repository\impl\PersonRepositoryImpl.java

```
public class PersonRepositoryImpl implements PersonRepositoryCustom {

    @PersistenceContext
    private EntityManager em;

    public List<Person> myQuery() {
        Query q = em.createQuery("from Person");
        return q.getResultList();
    }
}
```

使用 EntityManager 查询全部的 Person 数据。关于该接口的使用，可查阅 JPA 的相关文档。在此要特别说明的是，在默认情况下，Spring 会自动在包中寻找名为“PersonRepositoryImpl”的实现类，如果使用该类名找不到任何的类，则会将接口方法“myQuery”当作 Person 的字段进行数据查询，但 Person 类并没有 myQuery 字段，因此会抛出以下异常：

```
org.springframework.data.mapping.PropertyReferenceException: No property myQuery
found for type Person!
```

在这里要注意命名规则。运行 JpaApp，在浏览器中访问：http://localhost:8080/query，可以看到查询结果。

12.2.4 方法名查询

如果想根据 Person 的某个字段进行查询，实现类似“from Person where name = ?”这样的查询，则可以直接在 PersonRepository 接口中定义一个 name 方法，请见以下代码：

```
public interface PersonRepository extends JpaRepository<Person, Integer>,
        PersonRepositoryCustom {

    List<Person> name(String name);
}
```

Spring 会自动生成相应的查询方法，我们不需要编写任何实现逻辑。不仅如此，还可以使用接口的方法名，通过使用特定的关键字来实现查询功能，例如要查询 name 与 age 等于某值的 Person，则可以使用以下的方法名：

仅供非商业用途或交流学习使用


```
List<Person> findByNameAndAge(String name, Integer age);
```

又如，要查询 age 小于某值的 Person，可以使用以下方法名：

```
List<Person> findByAgeLessThan(Integer age);
```

Spring 通过定制方法名来实现相关的查询，其目前支持 20 多个关键字，如表 12-1 所示。

表 12-1 可以使用的关键字

关 键 字	例 子	对应的 SQL
And	findByLastnameAndFirstname	where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	where x.lastname = ?1 or x.firstname = ?2
Is,Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	where x.firstname = ?1
Between	findByStartDateBetween	where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	where x.age <= ?1
GreaterThan	findByAgeGreaterThan	where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	where x.age >= ?1
After	findByStartDateAfter	where x.startDate > ?1
Before	findByStartDateBefore	where x.startDate < ?1
IsNull	findByAgeIsNull	where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	where x.age not null
Like	findByFirstnameLike	where x.firstname like ?1
NotLike	findByFirstnameNotLike	where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	where x.firstname like ?1(parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	where x.firstname like ?1(parameter bound with prepended %)
Containing	findByFirstnameContaining	where x.firstname like ?1(parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	where x.age not in ?1
TRUE	findByActiveTrue()	where x.active = true
FALSE	findByActiveFalse()	where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	where UPPER(x.firstame) = UPPER(?1)

有了 Spring Data 提供的这些特性，在很多情况下，我们几乎不用编写代码，就可以完成查询功能。运行 JpaApp，在浏览器中访问：http://localhost:8080/nameage/员工 B/30，或者访问：http://localhost:8080/ageless/40，就可以看到具体的查询结果。

12.2.5 使用@Query 注解

如果使用方法名查询还是无法满足查询要求，你还可以在方法中使用@Query 注解，提供一段 JPQL（Java Presistence Query Language）语句或 SQL 语句，同样可以实现查询功能，请

仅供非商业用途或交流学习使用

见以下代码片断：

```
public interface PersonRepository extends JpaRepository<Person, Integer>,
        PersonRepositoryCustom {

    @Query("select p from Person p where p.name = ?1")
    List<Person> findPersonName(String name);
}
```

@Query 注解还支持使用原生的 SQL 查询：

```
@Query(value = "SELECT * FROM CRA_PERSON WHERE NAME = ?1", nativeQuery = true)
List<Person> findNativeByName(String name);
```

在接口方法中通过设置 nativeQuery 属性，可直接使用原生的 SQL 进行查询。运行 JpaApp，在浏览器中访问：http://localhost:8080/findPersonName/员工 A，或者访问：http://localhost:8080/native/员工 B，可以看到具体的查询效果。

除此之外，Spring Data JPA 模块还提供了很多其他的功能，提供这些功能的目的都是为了减少数据库开发的工作量，例如提供了分页查询、数据排序等这些常用的功能，这些功能涉及页面的功能开发，你可以到本书最后一章的案例中学习这方面的知识。

在以上的例子中，虽然在底层使用了 Hibernate 操作 MySQL 数据库，但我们基本上感觉不到 Hibernate 的存在，这也是 Spring Data 所希望达到的效果。下面，我们将使用同样的模型，来操作另外两个著名的数据库。

12.3 Spring Data 与 MongoDB

MongoDB 是一个开源的分布式数据库。经过多年的发展，MongoDB 已经相当稳定，目前，许多著名的公司都选择使用 MongoDB 作为数据存储的解决方案，例如 Facebook、Bosch 等。

我们首先讲述如何安装与配置 MongoDB，如果你已经安装了 MongoDB 环境，则可以跳过这部分内容。

12.3.1 安装 MongoDB

本书所使用的 MongoDB 版本为 3.4.9，可以到 MongoDB 官网下载，也可以到本书所附的 soft 目录中下载“mongodb-win32-x86_64-2008plus-ssl-3.4.9-signed.msi”。

MongoDB 默认会被安装在“C:\Program Files\MongoDB”目录下。使用命令行工具，进入“MongoDB\Server\3.4\bin”目录，输入命令：“mongod.exe --dbpath=D:\mongo_data\db”，其中—dbpath 用于指定数据目录。如果看到以下日志，则表示已经成功启动：

```
waiting for connections on port 27017
```

在默认情况下，不需要安全认证就可以直接进入 MongoDB。使用下面的步骤创建一个数据库：

1. 使用命令行工具，进入“MongoDB\Server\3.4\bin”目录。
2. 输入 mongo 命令。
3. 输入（创建）数据库命令：user crazyit。

仅供非商业用途或交流学习使用

④ 写入一条数据：db.runoob.insert({"name":"Angus"})。

⑤ 使用命令查看数据库：show dbs。

除了可以使用命令行来管理数据外，还可以使用一些可视化的管理工具，本书所附的 soft 目录中提供了“robo”管理软件，你可以下载使用。

12.3.2 配置权限

MongoDB 的用户与数据库绑定。可在命令行中建立用户并设置密码。使用 mongo 命令进入 MongoDB 后，输入以下命令新建用户：

```
db.createUser(
  {
    user: "test",
    pwd: "123456",
    roles: [ { role: "readWrite", db: "crazyit" } ]
  }
```

这里创建一个 test 用户，其拥有对 crazyit 的读写权限。创建用户后，还要开启 MongoDB 的权限认证，修改 MongoDB 服务器的启动命令：

```
mongod.exe --auth --dbpath=D:\mongo_data\db
```

测试用户是否可以登录。从命令行进入“MongoDB\Server\3.4\bin”，输入命令“mongo localhost/crazyit -utest -p123456”，随后可以看到，使用 test 用户成功进入 crazyit 数据库。

12.3.3 MongoDB 的相关概念

在使用 Spring Data MongoDB 模块时，会涉及 MongoDB 的一些基本概念。我们来简单了解一下。

- database：数据库，相当于 SQL 的数据库。
- collection：数据集合，相当于 SQL 的数据表。
- document：数据文档，可看作 SQL 的一条数据。
- field：数据域，可看作 SQL 中的 column。
- index：数据索引。

12.3.4 构建项目

与前面的 JPA 类似，Spring Boot 也提供相应的 Maven 依赖。新建“spring-mongodb”项目，在 pom.xml 中加入以下依赖（codes\12\spring-mongodb\pom.xml）：

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
```

仅供非商业用途或交流学习使用


```
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>
```

加入该依赖后，Spring Boot 会自动为我们的项目引入全部的依赖。主要的依赖包描述如下。

- spring-data-mongodb：Spring Data 的 MongoDB 模块。
- mongodb-driver：MongoDB 官方提供的驱动包。

新建 application.yml 文件，配置如下：

```
server:
  port: 8080
spring:
  data:
    mongodb:
      uri: mongodb://test:123456@localhost:27017/crazyit
```

这里使用 uri 属性来连接 MongoDB，格式为“mongodb://user:password@host:port/database”。除了 uri 属性外，还可以使用 host 和 port 配置连接。但要注意的是，MongoDB 3.0 以后的 Java 驱动，不再支持 spring.data.host 与 spring.data.port 配置，所以建议还是使用 uri 的方式进行配置。

按照之前使用 JPA 模块的方式，在项目中建立相应的包，来存放不同的源码，例如 controller、entity、repository、service。

12.3.5 数据访问层与业务层

首先在 crazyit 的数据库中，新建一个“Person”的 collection，创建命令为“db.createCollection(“Person”)”。新建 Person 对象，用于映射 collection，请见代码清单 12-6。

代码清单 12-6：codes\12\spring-mongodb\src\main\java\org\crazyit\mongo\entity\Person.java

```
@Document(collection="Person")
public class Person {

    @Id
    private String id;

    private String name;

    private Integer age;

    private String company;
    ...省略 setter 和 getter 方法
}
```

Person 使用@Document 注解进行修饰，指定映射“Person”的 collection。Person 实例与前面的 JPA 的类似，这里只是使用了不同的注解。接下来，编写数据访问接口，请见代码清单 12-7。

代码清单 12-7：codes\12\spring-mongodb\src\main\java\org\crazyit\mongo\repository\PersonRep.java

```
public interface PersonRep extends MongoRepository<Person, String> {

}
```

仅供非商业用途或交流学习使用

数据访问接口继承MongoRepository，不需要任何实现，它已经拥有最基本的CRUD操作。在业务层中，直接注入PersonRep实例，如代码清单12-8所示。

代码清单12-8：codes\12\spring-mongodb\src\main\java\org\crazyit\mongo\service\PersonService.java

```
@Service
public class PersonService {

    @Autowired
    private PersonRep personRep;

    public List<Person> getPersons() {
        List<Person> datas = personRep.findAll();
        return datas;
    }
}
```

除了findAll方法外，还有save、delete等方法，在此不一一列举。运行启动文件，本例为codes\12\spring-mongodb\src\main\java\org\crazyit\mongo\MongoApp.java。在浏览器中访问下面地址测试查询方法：http://localhost:8080/persons，访问下面地址测试新建方法：http://localhost:8080/save/myName。

12.3.6 自定义数据存储逻辑

Spring Data MongoDB模块同样支持自定义数据存储逻辑，就像JPA一样。新建自定义的存储接口，请见代码清单12-9。

代码清单12-9：codes\12\spring-mongodb\src\main\java\org\crazyit\mongo\repository\PersonRepCustom.java

```
public interface PersonRepCustom {

    List<Person> myQuery();
}
```

让原来的PersonRep接口同时继承MongoRepository和PersonRepCustom，再提供一个实现类，如代码清单12-10所示。

代码清单12-10：codes\12\spring-mongodb\src\main\java\org\crazyit\mongo\repository\impl\PersonRepImpl.java

```
public class PersonRepImpl implements PersonRepCustom {

    @Autowired
    private MongoTemplate mongoTemplate;

    public List<Person> myQuery() {
        List<Person> datas = mongoTemplate.execute(Person.class,
                new CollectionCallback<List<Person>>() {

                    public List<Person> doInCollection(MongoCollection collection)
                            throws MongoException, DataAccessException {
                        // 查询全部数据
                        FindIterable fi = collection.find();
                        MongoCursor<Document> cursor = fi.iterator();
                        List<Person> result = new ArrayList<Person>();
                        while (cursor.hasNext()) {
```

仅供非商业用途或交流学习使用


```
                    // 获取源数据实例
                    Document source = cursor.next();
                    // 转换为 Person
                    Person p = new Person();
                    ObjectId objectId = (ObjectId)source.get("_id");
                    p.setId(objectId.toHexString());
                    p.setAge((Integer) source.get("age"));
                    p.setName((String) source.get("name"));
                    p.setCompany((String) source.get("company"));
                    result.add(p);
                }
                return result;
            }
        });
        return datas;
    }
}
```

在自定义的 myQuery 方法中，使用 MongoTemplate 来执行数据查询。在调用 execute 方法时，实现一个 CollectionCallback 接口，这样的使用方式，接触过 Spring 与 Hibernate 的读者应该非常熟悉。在实现 CollectionCallback 接口时，使用 MongoDB 的 MongoCollection 对象进行数据查询，本例查询 MongoCollection 的全部数据，并封装为 Person 集合返回。

启动服务器（MongoApp.java），在浏览器中访问：http://localhost:8080/query，则可以看到自定义的数据查询结果。

12.3.7 方法名查询

JPA 与 MongoDB 模块在数据访问层、自定义数据存储逻辑实现上类似，除了这两个功能外，MongoDB 模块同样也支持根据方法名查询数据。在 PersonRep 接口中定义一个查询接口，不需要任何的实现，就可以实现根据 name 字段查询 Person 集合的功能：

```
List<Person> name(String name);
```

同样地，也可以使用以下的方法名，实现 Less Than 功能：

```
List<Person> findByAgeLessThan(Integer age);
```

除了以上的 LessThan 关键字外，方法名还支持表 12-2 中所列的关键字。

表 12-2 方法名关键字

关 键 字	例 子	对 应 逻 辑
After	findByBirthdateAfter(Date date)	{"birthdate" : {"$gt" : date}}
GreaterThan	findByAgeGreaterThan(int age)	{"age" : {"$gt" : age}}
GreaterThanEqual	findByAgeGreaterThanEqual(int age)	{"age" : {"$gte" : age}}
Before	findByBirthdateBefore(Date date)	{"birthdate" : {"$lt" : date}}
LessThan	findByAgeLessThan(int age)	{"age" : {"$lt" : age}}
LessThanEqual	findByAgeLessThanEqual(int age)	{"age" : {"$lte" : age}}
Between	findByAgeBetween(int from, int to)	{"age" : {"$gt" : from, "$lt" : to}}
In	findByAgeIn(Collection ages)	{"age" : {"$in" : [ages…]}}
NotIn	findByAgeNotIn(Collection ages)	{"age" : {"$nin" : [ages…]}}

仅供非商业用途或交流学习使用

续表

关 键 字	例 子	对 应 逻 辑
IsNotNull, NotNull	findByFirstnameNotNull()	{"firstname" : {"$ne" : null}}
IsNull, Null	findByFirstnameNull()	{"firstname" : null}
Like, StartingWith, EndingWith	findByFirstnameLike(String name)	{"firstname" : name} (name as regex)
NotLike, IsNotLike	findByFirstnameNotLike(String name)	{"firstname" : { "$not" : name }} (name as regex)
Containing on String	findByFirstnameContaining(String name)	{"firstname" : name}
NotContaining on String	findByFirstnameNotContaining(String name)	{"firstname" : { "$not" : name}}
Containing on Collection	findByAddressesContaining(Address address)	{"addresses" : { "$in" : address}}
NotContaining on Collection	findByAddressesNotContaining(Address address)	{"addresses" : { "$not" : { "$in" : address}}}
Regex	findByFirstnameRegex(String firstname)	{"firstname" : {"$regex" : firstname }}
Not	findByFirstnameNot(String name)	{"firstname" : {"$ne" : name}}
Near	findByLocationNear(Point point)	{"location" : {"$near" : [x,y]}}
Near	findByLocationNear(Point point, Distance max)	{"location" : {"$near" : [x,y], "$maxDistance" : max}}
Near	findByLocationNear(Point point, Distance min, Distance max)	{"location" : {"$near" : [x,y], "$minDistance" : min, "$maxDistance" : max}}
Within	findByLocationWithin(Circle circle)	{"location" : {"$geoWithin" : {"$center" : [[x, y], distance]}}}
Within	findByLocationWithin(Box box)	{"location" : {"$geoWithin" : {"$box" : [[x1, y1], x2, y2]}}}
IsTrue, True	findByActiveIsTrue()	{"active" : true}
IsFalse, False	findByActiveIsFalse()	{"active" : false}
Exists	findByLocationExists(boolean exists)	{"location": {"$exists" : exists }}

其中，“$gt”、“$gte”、“$lt”、“$lte”为 MongoDB 的查询操作符，关于这些操作符的作用，在此不展开讲解，读者可查阅 MongoDB 的相关文档了解。

在数据访问层接口的方法名中使用这些关键字，可以帮我们实现各种查询功能，一方面可以减轻编码压力，另一方面也减少出错的概率。

启动服务器（MongoApp.java），在浏览器中访问：http://localhost:8080/name/myName，根据名称值“myName”来查询数据，访问“http://localhost:8080/ageless/34”进行 LessThan 查询，这样将会查询出 age 值小于 34 的数据。

12.3.8 使用@Query 注解

如果定义接口无法满足查询需求，则可以使用@Query 注解来定义查询条件，请见以下代码片断：

```
@Query("{ 'name' : ?0, 'age' : ?1 }")
List<Person> findByNameAndAge(String name, Integer age);
```

仅供非商业用途或交流学习使用

为@Queruy 注解传入用于查询的 JSON 语句即可。除了查询外，还可以定制查询的字段，如以下代码片断：

```
@Query(value = "{ 'name' : ?0 }", fields = "{ 'name' : 1, 'company' : 1}")
List<Person> findByName(String name);
```

以上的@Query 注解添加了 fields 字段，表示最终查询出来的 Person 只有 id、name、company 属性，其他的属性值为 null。

启动服务器（MongoApp.java），在浏览器中访问“http://localhost:8080/nameage/myName/33”，根据 name 与 age 查询数据。

注意：虽然都是@Query 注解，而且使用方式类似，但是这个@Query 注解与 JPA 的@Query 注解是两个完全不同的注解，读者在使用时请勿混淆。

Spring Data MongoDB 模块还有很多强大的功能，限于篇幅，不一一列举了。下面，讲述 Spring Data 与 Redis。

12.4 Spring Data 与 Redis

Redis 是一款开源的 key-value 数据库，除了 key-value 外，它还支持 set、hash 等数据结构。目前可供选择的 Redis 客户端有很多，在 Java 领域，使用较为广泛的客户端有 Redisson、Jedis，而 Spring Data Redis 模块默认使用 Jedis。

12.4.1 Redis 的安装与配置

Redis 的正式版不支持 Windows，但是在 Git 上面，可以找到一个 Windows 版本的 Redis 服务器（只支持 64 位的 Windows），本小节将使用其作为 Redis 服务器，读者可以从本书的 soft 目录下载 Redis-x64-3.2.100.zip 压缩包。

将下载的 Redis 服务器压缩包解压到具体目录，并使用命令行工具进入目录，运行“redis-server.exe redis.windows.conf”，Redis 服务器启动画面如图 12-3 所示。



图 12-3 启动 Redis 服务器

仅供非商业用途或交流学习使用

默认占用 6379 端口。在默认情况不需要输入密码即可进入 Redis，可以在配置文件“redis.windows.conf”中，修改或添加属性：“requirepass 123456”。如果需要修改连接端口，则可以修改该配置文件中的 port 属性，修改完配置后需要重启 Redis 服务器。

启动 Redis 服务器后，使用命令行工具，进入 Redis 目录，执行“redis-cli.ex –a 123456”命令，即可登录到 Redis。在 Redis 客户端的命令行中，输入“set name_a test_a”，表示往 Redis 中写一条 key 为“name_a”的 string 类型数据，再使用“get name_a”获取该数据值。

目前有许多图形化的 Redis 管理工具，读者可以选择适合自己的来使用。本书的 soft 目录中提供了一个名为“RedisStudio”的管理软件，你可以下载使用。

12.4.2 Redis 的数据类型

Redis 使用 key-value 的结构来保存数据，其中 value 支持 5 种数据类型：string、hash、list、set、zset。下面对这 5 种数据类型做一个简单的描述。

- string：最基本的数据类型，可保存任何的数据。
- hash：一个键值对的集合，在集合中以字段名作为 key，字段值作为 value，主要用于保存对象。
- list：字符串列表，可以往列表中添加元素。
- set：无序的集合，集合内数据不能重复。
- zset：有序的集合，集合内数据不能重复。

在进行数据存储时，要注意数据类型，例如保存时使用的是 hash 类型，那么在获取数据时，也要使用对应的类型。

12.4.3 使用 Jedis

如果需要保存一些不会重复的数据（例如 UUID），使用 set 类型来保存较为理想；如果保存的是 Java 对象，则使用 hash 类型保存较为合适。Spring Data 默认使用 Jedis 来操作 Redis。我们先看下 Jedis 如何操作 hash 与 set 类型的数据。新建一个名为“test-jedis”的 Maven 项目，为其加入 Jedis 的依赖：

```
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>2.9.0</version>
        </dependency>
```

分别读写 hash 与 set 类型的数据，请见代码清单 12-11。

代码清单 12-11：codes\12\test-jedis\src\main\java\org\crazyit\cloud\JetisTest.java

```
public class JetisTest {

    public static void main(String[] args) {
        // 登录本地的 Redis
        Jedis jedis = new Jedis("localhost");
        jedis.auth("123456");
        // 将数据保存为 hash 类型
        Map<String, String> data = new HashMap<String, String>();
        data.put("name", "Angus");
```

仅供非商业用途或交流学习使用


```
        data.put("age", "33");
        data.put("company", "crazyit");
        jedis.hmset("person_test", data);                                    ①
        // 查询 hash 类型的数据
        List<String> dbDatas = jedis.hmget("person_test", "name", "age");    ②
        for(String dbData : dbDatas) {
            System.out.println(dbData);
        }
        /* =========分隔线==========  */
        // 将数据保存为 set 类型
        jedis.sadd("person_test_ids", "1", "2");                             ③
        // 查询数据
        Set<String> dbDatasSet = jedis.smembers("person_test_ids");          ④
        for(String dbData : dbDatasSet) {
            System.out.println(dbData);
        }
    }
}
```

①和②处的代码分别写入和读取 hash 类型的数据，③和④处的代码分别写入和读取 set 类型的数据。运行代码清单 12-11 后，使用 RedisStudio 工具查看添加的数据，如图 12-4 与图 12-5 所示。

Key: person_test
Type: hash
Size: 3
TTL: -1

Field	Value
company	crazyit
age	33
name	Angus

图 12-4 hash 数据

Key: person_test_ids
Type: set
Size: 2
TTL: -1

Value
1
2

图 12-5 set 数据

12.4.4 构建 Spring Data 项目

Spring Boot 同样提供了 Spring Data 与 Redis 的依赖。新建名称为“spring-redis”的 Maven 项目，在 pom.xml（codes\12\spring-redis\pom.xml）中加入以下依赖：

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

仅供非商业用途或交流学习使用


```
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>
```

新建 application.yml 文件，内容如下：

```
server:
  port: 8080
spring:
  redis:
    port: 6379
    password: 123456
    host: localhost
```

使用 spring.redis.x 来配置 Redis 的相关属性。除了以上配置方法外，还可以使用 spring.redis.url 属性来配置，格式为：redis://user:password@host:port，其中可以随便填写 user，它会被忽略。该配置等价于以下配置：

```
url: redis://ignore:123456@localhost:6379
```

与前面的 JPA、MongoDB 中的例子一样，为项目建立相应的包：controller、repository、service、entity。

12.4.5 数据访问层与业务层

新建一个 Person 类，用于映射 Redis 的数据，请见代码清单 12-12。

代码清单 12-12：codes\12\spring-redis\src\main\java\org\crazyit\redis\entity\Person.java

```
@RedisHash("Person")
public class Person {

    @Id
    private String id;

    @Indexed
    private String name;

    private Integer age;

    private String company;
    ...省略 setter 和 getter 方法
}
```

在 Person 类中，使用@RedisHash 注解，表示 Person 对象将会以 hash 类型被保存。实际上，在实现时，Person 数据的 id，将会作为 set 类型被保存。还需要注意 Person 类的 name 属性，该属性使用了@Indexed 注解，表示在保存 Person 时，会对 name 属性建立索引。

接下来，编写数据访问接口，请见代码清单 12-13。

代码清单 12-13：codes\12\spring-redis\src\main\java\org\crazyit\redis\repository\PersonRep.java

```
public interface PersonRep extends CrudRepository<Person, String> {

}
```

数据访问接口继承了 CrudRepository，同样不需要任何的实现，PersonRep 接口即拥有普通的 CRUD 操作。编写业务类，添加保存与查询方法，请见代码清单 12-14。

代码清单 12-14：codes\12\spring-redis\src\main\java\org\crazyit\redis\service\PersonService.java

```
@Service
public class PersonService {

    @Autowired
    private PersonRep personRep;

    public List<Person> getPersons() {
        Iterable<Person> persons = personRep.findAll();
        List<Person> datas = new ArrayList<Person>();
        for(Iterator<Person> it = persons.iterator(); it.hasNext();) {
            Person p = it.next();
            datas.add(p);
        }
        return datas;
    }

    public void save(String name) {
        Person p = new Person();
        p.setName(name);
        p.setAge(33);
        p.setCompany("crazyit");
        personRep.save(p);
    }
}
```

在代码清单 12-14 的粗体代码中，调用了查找和保存方法。注意，CrudRepository 接口的 findAll 方法返回的是 Iterable 实例，需要对其遍历并将 Person 放到 List 中。为了查看保存到 Redis 中的数据类型，可调用 save 方法来保存 Person。

保存两次 Person，使用 RedisStudio 查看，可以看到数据如图 12-6 和图 12-7 所示。

可以看到，所有 Person 的 id 都使用 set 类型进行保存，而具体的 Person 数据，则会使用 hash 类型进行保存。一个 Person 实例的 key 格式为：“Person:UUID”。

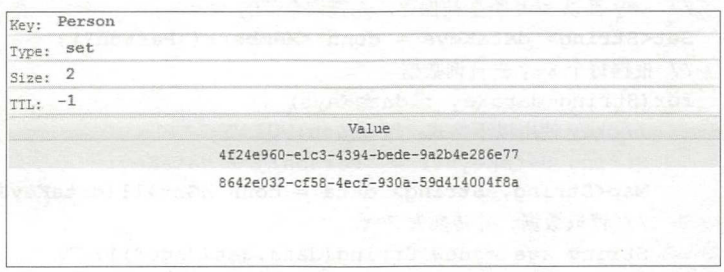


图 12-6 保存的 id

Key: Person:4f24e960-e1c3-4394-bede-9a2b4e286e77
Type: hash
Size: 5
TTL: -1

Field	Value
_class	org.crazyit.redis.entity.Person
id	4f24e960-e1c3-4394-bede-9a2b4e286e77
name	test2
age	33
company	crazyit

图 12-7　保存的数据

启动服务器，本例为 codes\12\spring-redis\src\main\java\org\crazyit\redis\RedisApp.java。在浏览器中访问 http://localhost:8080/persons，可看到查询结果。访问 http://localhost:8080/save/myName，添加名称为“myName”的 Person 数据。

12.4.6　自定义数据存储逻辑

新建一个 PersonRepoCustom 接口，用于自定义数据存储接口，请见代码清单 12-15。

代码清单 12-15：codes\12\spring-redis\src\main\java\org\crazyit\redis\repository\PersonRepCustom.java

```
public interface PersonRepCustom {

    List<Person> myQuery();
}
```

让原来的 PersonRep 接口同时继承 CrudRepository 和 PersonRepCustom，新建 PersonRepCustom 接口的实现类，如代码清单 12-16 所示。

代码清单 12-16：codes\12\spring-redis\src\main\java\org\crazyit\redis\repository\impl\PersonRepImpl.java

```
public class PersonRepImpl implements PersonRepCustom {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    public List<Person> myQuery() {
        List<Person> datas = stringRedisTemplate.execute(new RedisCallback<List<Person>>() {

            public List<Person> doInRedis(RedisConnection connection)
                    throws DataAccessException {
                List<Person> result = new ArrayList<Person>();
                StringRedisConnection conn = (StringRedisConnection)connection;
                // key 都以 set 类型被保存，先查询全部的 key
                Set<String> dataKeys = conn.sMembers("Person");
                // 根据每个 key 去查询数据
                for(String dataKey : dataKeys) {
                    // key 使用以下格式："Person:UUID"
                    String dataKeyStr = "Person:" + dataKey;
                    Map<String, String> data = conn.hGetAll(dataKeyStr);
                    // 读取数据，并转换为 Person
                    String age = new String(data.get("age"));
                    String name = new String(data.get("name"));
                    String company = new String(data.get("company"));
                    String id = new String(data.get("id"));
                    Person p = new Person();
```

```
                p.setId(id);
                p.setAge(Integer.parseInt(age));
                p.setName(name);
                p.setCompany(company);
                result.add(p);
            }
            return result;
        }
    });
    return datas;
  }
}
```

在代码清单 12-16 中，向数据访问层注入了 StringRedisTemplate。与前面 MongoDB 中的例子类似，实现了 RedisCallback 接口，并在 doInRedis 方法中，查询了全部的 Person 数据。

前面讲过，Person 的 id 会以 set 类型被保存，而 Person 会以 hash 类型被保存，因此在实现具体的查询逻辑时，要先查询全部的 id，然后再根据 id 去查询每一个 Person。注意，在查询具体的 Person 时，Person 数据的 key 格式为：“Person:UUID”。

代码清单 12-6 中的查询过程，实际上 Spring Data 已经帮我们实现了，这里只是将该过程重新展现而已。运行服务器（RedisApp.java），在浏览器中访问 http://localhost:8080/query，即可以看到自定义的数据查询结果。

注意：在命名实现类时，要注意命名规则，即原存储接口名称加上“Impl”。

12.4.7 方法名查询

Redis 模块同样支持方法名查询，例如要根据 Person 的 name 来查询，可定义以下的接口方法：

```
List<Person> findByName(String name);
```

注意：使用这种方式进行查询，必须要在 name 字段建立索引。

方法名还可以使用以下的关键字。

- And：例如，在接口中可以定义“findByNameAndAge”。
- Or：例如，“findByNameOrAge”。
- Is、Equals：例如，“findByNameIs”、“findByName”、“findByNameEquals”。
- Top、First：例如，“findFirst5Name”、“findTop5ByName”，查询前 5 条记录。

除了本章介绍的功能外，Spring Data Redis 模块还有其他的功能，由于篇幅所限，不再列举。

12.5 本章小结

本章主要讲述了如何以 Spring Data 为基础，在 Spring Boot 中开发数据库应用。具体介绍了如何使用 Spring Data 操作 MySQL、MongoDB 和 Redis。Spring Data 为我们提供了一种统一的编码风格，让我们几乎可以使用同样的模型来开发数据库应用。本章的内容不涉及前端页面的功能开发，在后面的章节中，我们会结合 Thyemeleaf、Spring Data 等框架来开发最基本的系统功能。

CHAPTER

13

第13章 开发微服务

本章要点

- 微服务与 Spring Cloud
- Eureka 服务治理
- Ribbon 负载均衡
- 客户端 Feign
- 容错框架 Hystrix

本章将简述 Spring Cloud 的几个主要架构，学习完本章，你可以开发最基本的微服务应用。

13.1 微服务与 Spring Cloud

13.1.1 微服务简介

“微服务”一词来源于 Martin Fowler 的“Microservices”一文。微服务是一种架构模式，在这种架构模式中，将单体应用划分为小型的服务单元，微服务之间使用 HTTP 的 API 进行资源访问与操作。

在对单体应用的划分上，微服务与 SOA 架构有点类似，但是 SOA 架构侧重于将每个单体应用的服务集成到 ESB 上，而微服务做得更加彻底，强调将整个模块变成服务组件，微服务对模块的划分粒度可能会更细。SOA 架构与微服务架构分别如图 13-1 与图 13-2 所示。

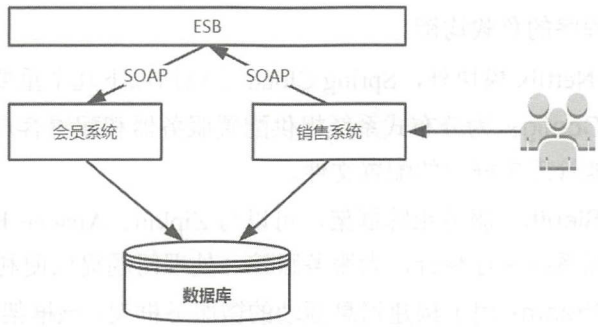


图 13-1 SOA 架构

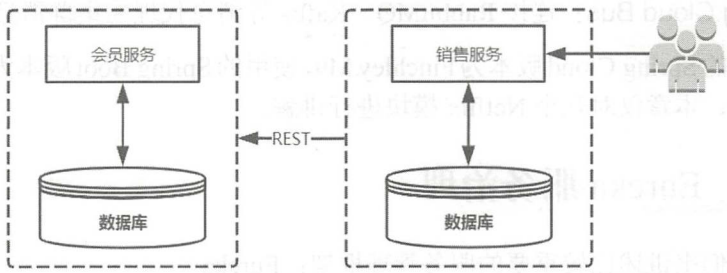


图 13-2 微服务架构

Martin Fowler 在文章中肯定了 Netflix 在微服务架构方面的贡献，下面，我们来了解一下 Netflix。

13.1.2 Netflix 与 Spring Cloud

Netflix 是一个互联网影片提供商，在几年前，Netflix 公司成立了自己的开源中心，名称为 Netflix Open Source Software Center，简称 Netflix OSS。这个开源组织专注于大数据、云计算方面的技术，提供了多个开源框架，包括大数据工具、构建工具、基于云平台的服务工具等。Netflix 所提供的这些框架，很好地遵循了微服务所推崇的理念，实现了去中心化的服务管理、服务容错等机制。

Spring Cloud 并不是一个具体的框架，大家可以把它理解为一个工具箱，它提供的各类工

具，可以帮助我们快速构建分布式系统。Spring Cloud 的各个项目基于 Spring Boot，将 Netflix 的多个框架进行封装，并且通过自动配置的方式将这些框架绑定到 Spring 的环境中，从而使得使用这些框架变得非常简单。由于 Spring Boot 的这种简便性，使得我们在使用 Spring Cloud 时，很容易将 Netflix 各个框架整合进我们的项目中。

➤➤ 13.1.3 Spring Cloud 模块介绍

Spring Cloud 下的“Spring Cloud Netflix”模块，主要封装了 Netflix 的以下项目。

- Eureka：基于 REST 服务的分布式中间件，主要用于服务管理。
- Hystrix：容错框架，通过添加延迟阈值以及容错逻辑，来帮助我们控制分布式系统间组件的交互。
- Feign：一个 REST 客户端，目的是为了简化 Web Service 客户端的开发。
- Ribbon：负载均衡框架，其在微服务集群中为各个客户端的通信提供支持，它主要实现中间层应用程序的负载均衡。

除了 Spring Cloud Netflix 模块外，Spring Cloud 还包括以下几个重要的模块。

- Spring Cloud Config：为分布式系统提供配置服务器和配置客户端，通过对它们的配置，可以很好地管理集群中的配置文件。
- Spring Cloud Sleuth：服务跟踪框架，可以与 Zipkin、Apache HTrace 和 ELK 等数据分析、服务跟踪系统进行整合，为服务跟踪、处理问题提供便利。
- Spring Cloud Stream：用于构建消息驱动的微服务框架，该框架在 Spring Boot 的基础上，整合了“Spring Integration”来连接消息代理中间件。
- Spring Cloud Bus：连接 RabbitMQ、Kafka 等消息代理的集群消息总线。

本章使用的 Spring Cloud 版本为 Finchley.M9，使用的 Spring Boot 版本为 2.0.0. RELEASE。由于篇幅所限，本章仅对几个 Netflix 模块进行讲解。

13.2 Eureka 服务治理

这一节我们来讲述比较重要的服务管理框架：Eureka。

➤➤ 13.2.1 Eureka 架构

Eureka 提供基于 REST 的服务，在集群中主要用于服务管理。Eureka 提供了基于 Java 语言的客户端组件，客户端组件实现了负载均衡的功能，这为业务组件的集群部署创造了条件。使用该框架，可以将业务组件注册到 Eureka 容器中，这些业务组件可进行集群部署，Eureka 主要维护这些服务的列表并自动检查它们的状态。图 13-3 所示是一个 Eureka 服务集群的架构。

图 13-3 中有两个服务器，服务器支持集群部署，每个服务器也可以作为对方服务器的客户端进行相互注册与复制。三个 Eureka 客户端，两个用于发布服务，一个用于调用服务。图中的每一个组件，都可以作为一个独立的 Spring Boot 应用。不管是在服务器上还是在客户端上，都可以部署多个实例，如此一来，就很容易构建高可用的服务集群。下面，我们构建一个简单的微服务集群。

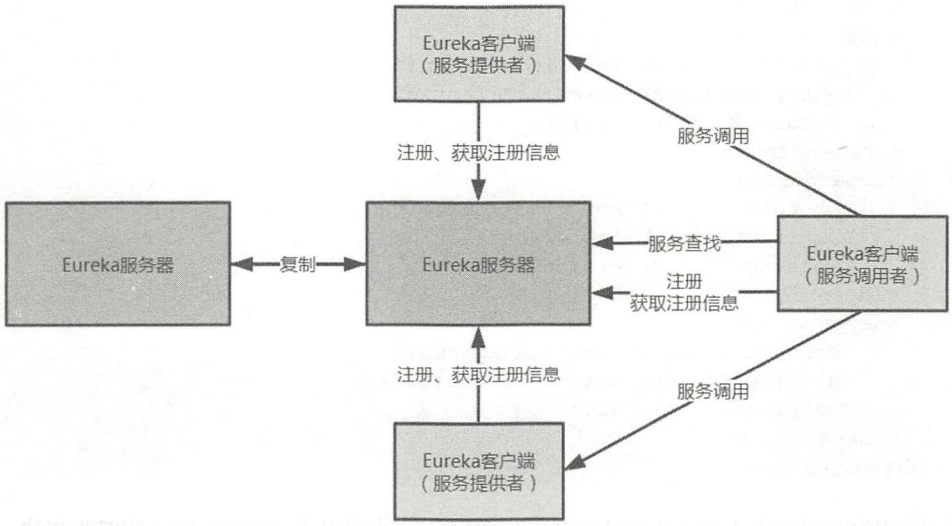


图 13-3 Eureka 架构

➤➤ 13.2.2 构建服务器

先创建一个名称为“first-ek-server”的 Maven 项目作为服务器，并在 pom.xml 文件中加入 Spring Cloud 的依赖，如代码清单 13-1 所示。

代码清单 13-1：codes\13\13.2\first-ek-server\pom.xml

```
<!-- 声明项目的父项目 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>
<!-- 添加依赖管理 -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Finchley.M9</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <!-- 引用 Eureka Server 依赖 -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>
    <!-- 使用 JDK10 需要加入以下几个依赖 -->
    <dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-core</artifactId>
            <version>2.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-impl</artifactId>
            <version>2.3.0</version>
        </dependency>
        <dependency>
            <groupId>javax.activation</groupId>
            <artifactId>activation</artifactId>
            <version>1.1</version>
        </dependency>
    </dependencies>
```

加入的 spring-cloud-starter-netflix-eureka-server 会自动引入 spring-boot-starter-web，因此只需要加入该依赖，我们的项目就具有了 Web 容器的功能，这也是 Spring Boot 的优点之一，即引入依赖，一站式建立项目。接下来，编写一个最简单的启动类，来启动我们的 Eureka 服务器，启动类如代码清单 13-2 所示。

代码清单 13-2：codes\13\13.2\first-ek-server\src\main\java\org\crazyit\cloud\FirstServer.java

```
@SpringBootApplication
@EnableEurekaServer
public class FirstServer {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FirstServer.class).run(args);
    }
}
```

该启动类几乎与前面章节中的 Spring Boot 项目的一致，只是加入了@EnableEurekaServer 注解，声明这是一个 Eureka 服务器。直接运行 FirstServer 即可启动 Eureka 服务器。需要注意的是，本例并没有配置服务器端口，其默认端口为 8080，我们在此将端口配置为 8761，因而在 src/main/resources 目录下创建 application.yml 配置文件，内容如下：

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

其中的 eureka.client.registerWithEureka 属性，声明是否将自己的信息注册到 Eureka 服务器中，默认值为 true。eureka.client.fetchRegistry 属性则表示，是否到 Eureka 服务器中抓取注册信息。将这两个属性的值设置为 false，则启动时不会出现异常信息。运行启动类，在浏览器中访问：http://localhost:8761，浏览器显示如图 13-4 所示，表示 Eureka 启动成功。

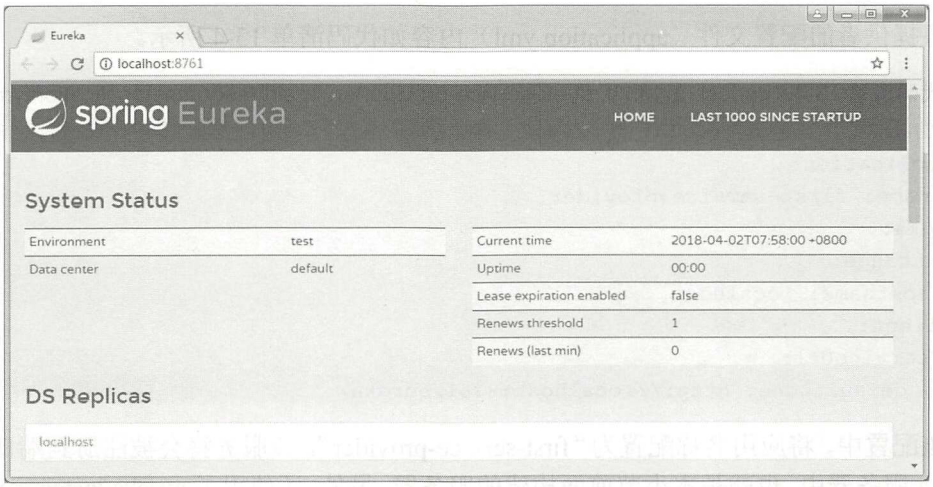


图 13-4 Eureka 控制台

➤➤ 13.2.3 服务提供者

服务提供者，就像 WebService 的服务端一样，提供接口服务让外部调用。建立名称为“first-ek-service-provider”的项目，在 pom.xml 中加入依赖，如代码清单 13-3 所示。

代码清单 13-3：codes\13\13.2\first-ek-service-provider\pom.xml

```
<!-- 声明项目的父项目 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>
<!-- 添加依赖管理 -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Finchley.M9</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>
```

服务提供者的配置文件（application.yml）内容如代码清单 13-4 所示。

代码清单 13-4：codes\13\13.2\first-ek-service-provider\src\main\resources\application.yml

```
spring:
  application:
    name: first-service-provider
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

在该配置中，将应用名称配置为“first-service-provider”，该服务将会被注册到端口为 8761 的 Eureka 服务器中，也就是本小节前面构建的服务器。另外，还使用了 eureka.instance.hostname 来配置该服务实例的主机名称。编写一个 Controller 类，并提供一个最简单的 REST 服务，本例将 Controller 类与启动类放在同一个类中，请见代码清单 13-5。

代码清单 13-5：codes\13\13.2\first-ek-service-provider\src\main\java\org\crazyit\cloud\FirstServiceProvider.java

```
@SpringBootApplication
@EnableEurekaClient
@RestController
public class FirstServiceProvider {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FirstServiceProvider.class).run(args);
    }

    @GetMapping(value = "/person/{personId}", produces = MediaType.APPLICATION_
JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId) {
        Person person = new Person(personId, "Crazyit", 30);
        return person;
    }
}
```

在启动类中，使用了@EnableEurekaClient 注解，声明该应用是一个 Eureka 客户端。发布了一个“/person/{personId}”的服务，返回 JSON 字符串。启动服务提供者（FirstServiceProvider.java），在浏览器中访问：http://localhost:8080/person/1，则可以看到服务接口。

先启动服务器（代码清单 13-2，8761 端口），再启动服务提供者（代码清单 13-5，8080 端口），在浏览器中访问 Eureka 控制台：http://localhost:8761，结果如图 13-5 所示。

从图 13-5 可以看到，当前注册的服务列表中，只有我们编写的“first-service-provider”，这表明服务注册成功。接下来编写服务调用者。

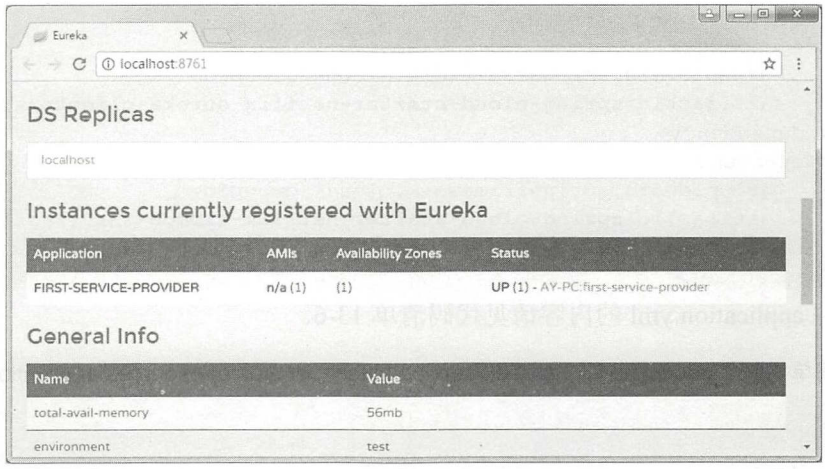


图 13-5 查看发布的服务

➤➤ 13.2.4 服务调用者

服务被注册、发布到 Eureka 服务器后，需要提供给外部调用，如果没有 Eureka，我们一般直接通过 Web Service 来调用，但有了 Eureka 后，我们可以通过 Eureka 来管理这些服务。

此处所说的调用者，是指同样被注册到 Eureka 的客户端，它调用其他客户端发布的服务。由于同一个服务，可能会部署多个实例，因此调用过程就可能涉及负载均衡、服务器查找等问题，所幸 Netflix 的项目可以帮我们解决这类问题，并且 Spring Cloud 已经封装了一次，因此我们仅需编写少量代码，就可以实现服务调用。

新建名称为“first-ek-service-invoker”的项目，在 pom.xml 文件中加入以下依赖：

```
<!-- 声明项目的父项目 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>
<!-- 添加依赖管理 -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Finchley.M9</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
        </dependency>
    </dependencies>
```

配置文件 application.yml 的内容请见代码清单 13-6。

代码清单 13-6：codes\13\13.2\first-ek-service-invoker\src\main\resources\application.yml

```
server:
  port: 9000
spring:
  application:
    name: first-service-invoker
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

在代码清单 13-6 中，配置应用的名称为 first-service-invoker，该应用会被作为 Eureka 客户端注册到本地的 8761 端口。编写控制器与启动类，让调用者对外提供一个测试服务，代码清单 13-7 为控制器的代码实现。

代码清单 13-7：codes\13\13.2\first-ek-service-invoker\src\main\java\org\crazyit\cloud\FirstInvoker.java

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class FirstInvoker {

    public static void main(String[] args) {
        SpringApplication.run(FirstInvoker.class, args);
    }

    @Bean
    @LoadBalanced
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }

    @GetMapping(value = "/router", produces = MediaType.APPLICATION_JSON_VALUE)
    public String router() {
        RestTemplate restTpl = getRestTemplate();
        // 根据应用名称调用服务
        String json = restTpl.getForObject(
                "http://first-service-provider/person/1", String.class);
        return json;
    }
}
```

在控制器中，配置了 RestTemplate 的 bean，RestTemplate 本来是 spring-web 模块下面的类，

主要用来调用 REST 服务，其本身并不具备调用分布式服务的能力，但是 RestTemplate 的 bean 被@LoadBalanced 注解修饰后，RestTemplate 实例就具有了访问分布式服务的能力。

在控制器中，新建了一个 router 的测试方法，用来对外发布 REST 服务。该方法只起一个路由的作用，实际上使用 RestTemplate 来调用“first-ek-service-provider”（服务提供者）的服务。需要注意的是，在调用服务时，仅仅通过服务名称进行调用。RestTemplate 会根据服务 id 调用具体服务，至于服务由哪个服务器提供，客户端并不需要关心。按照以下步骤测试程序。

1. 启动 Eureka 服务器：代码清单 13-2，8761 端口。
2. 启动服务提供者：代码清单 13-5，8080 端口。
3. 启动服务调用者：代码清单 13-7，9090 端口。

在浏览器中访问：http://localhost:9000/router，结果如下：

```
{"id":1,"name":"Crazyit","age":30}
```

根据输出可知，实际上调用了服务提供者的/person/1 服务。第一个 Eureka 应用就介绍到这里，下面，我们介绍一下这个程序的结构。

➤➤ 13.2.5 程序结构

本示例新建了三个项目，如果读者对程序的结构不太清楚，可以参看图 13-6。

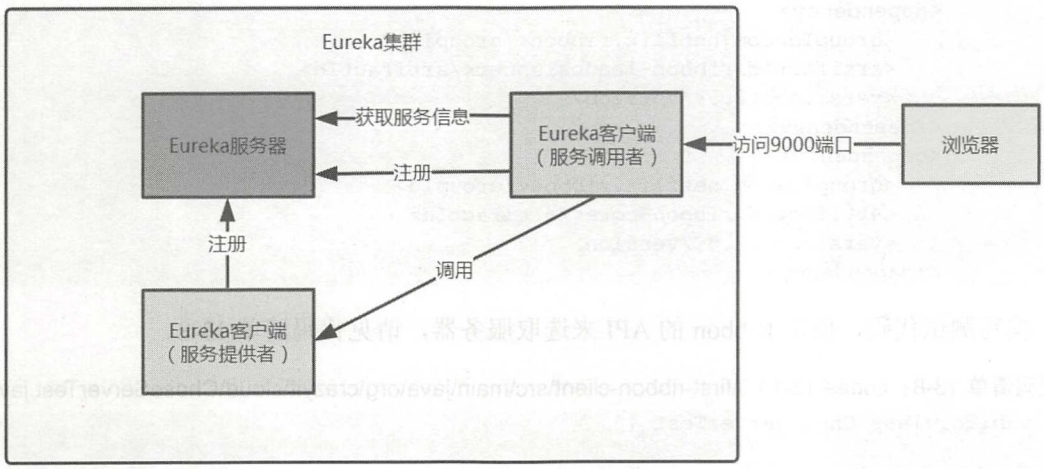


图 13-6 程序结构

Eureka 服务器为本例的“first-ek-server”，服务提供者为“first-ek-service-provider”，而调用者为“first-ek-service-invoker”。用户通过浏览器访问调用者的 9000 端口的 router 服务，在 router 服务中查找服务提供者的服务并进行调用。在本例中，服务调用有点像路由器路由，对于这种结构，我们可以为服务提供者启动多个实例（从不同的端口），将每个实例都注册到 Eureka 服务器中，而对于服务调用者，只需要知道要调用的服务 id 即可，至于具体由哪个实例提供服务，服务调用者并不需要关心，这样就可以实现负载均衡。

13.3 Ribbon 负载均衡

负载均衡是分布式架构关注的重点，负载均衡机制将决定着整个服务集群的性能与稳定性。

如果服务提供者部署了多个实例，每个实例都可以平均处理服务请求，那么这些请求是如何被分摊到各个服务实例中的呢？我们下面就来学习负载均衡框架 Ribbon。

➤➤ 13.3.1 Ribbon 介绍

Ribbon 是 Netflix 下的负载均衡项目，在集群中为各个客户端的通信提供支持，它主要实现中间层应用程序的负载均衡。Ribbon 提供以下特性。

- 负载均衡器，可支持插拔式的负载均衡规则。
- 对多种协议提供支持，例如 HTTP、TCP、UDP 等。
- 集成负载均衡功能的客户端。

同为 Netflix 项目，Ribbon 可以与 Eureka 整合使用，Ribbon 同样可以被集成到 Spring Cloud 中，作为 spring-cloud-netflix 项目中的子模块。Spring Cloud 将 Ribbon 的 API 进行了封装，你可以使用封装后的 API 来实现负载均衡，也可以直接使用 Ribbon 的原生 API。

➤➤ 13.3.2 第一个 Ribbon 程序

Ribbon 的负载均衡器接口定义了服务器的操作，主要是进行服务器选择。大家只要记住一点，Ribbon 是从一堆服务器中选取一个服务器出来，至于选取出来的服务器做什么工作，Ribbon 并不关心，它只是选取服务器。新建一个 Maven 项目，在 pom.xml 中加入以下依赖：

```
        <dependency>
            <groupId>com.netflix.ribbon</groupId>
            <artifactId>ribbon-loadbalancer</artifactId>
            <version>2.2.5</version>
        </dependency>
        <dependency>
            <groupId>com.netflix.ribbon</groupId>
            <artifactId>ribbon-core</artifactId>
            <version>2.2.5</version>
        </dependency>
```

编写测试代码，使用 Ribbon 的 API 来选取服务器，请见代码清单 13-8。

代码清单 13-8：codes\13\13.3\first-ribbon-client\src\main\java\org\crazyit\cloud\ChoseServerTest.java

```
public class ChoseServerTest {

    public static void main(String[] args) {
        // 创建负载均衡器
        BaseLoadBalancer lb = new BaseLoadBalancer();
        // 添加服务器
        List<Server> servers = new ArrayList<Server>();
        servers.add(new Server("localhost", 8080));
        servers.add(new Server("localhost", 8081));
        lb.addServers(servers);
        // 进行 6 次服务器选择
        for(int i = 0; i < 6; i++) {
            Server s = lb.chooseServer(null);
            System.out.println(s);
        }
    }
}
```

在该代码中使用了 BaseLoadBalancer 这个负载均衡器，并将两个服务器对象加入到了负载均衡器中，然后调用了 6 次 chooseServer 方法。运行代码清单 13-8，输出如下：

```
localhost:8081
localhost:8080
localhost:8081
localhost:8080
localhost:8081
localhost:8080
```

可以看到，每个 Server 都被选取一次，至于选取的服务器用来做什么，并不是 Ribbon 所要关心的。很多时候，我们会让这个选中的服务器去处理请求。

➤➤ 13.3.3 在 Spring Cloud 中测试使用 Ribbon

下面，我们在 Spring Cloud 中整合使用 Ribbon。实际上在前面章节中，我们就已经接触过 Ribbon，在前面实现服务调用者时，使用@LoadBalanced 注解来修饰 RestTemplate，此时 RestTemplate 就具有负载均衡的功能。下面我们编写一个简单的例子来进行测试。

分别新建 Eureka 服务器、服务提供者与服务调用者项目，请参考 12.2 节的内容，本例中三个项目的路径如下。

- **Eureka 服务器：** codes\13\13.3\ek-server
- **服务提供者：** codes\13\13.3\ek-service-provider
- **服务调用者：** codes\13\13.3\ek-service-invoker

为了测试 Ribbon 的作用，我们需要修改服务提供者，并对启动类与控制器做部分修改，请见代码清单 13-9。

代码清单 13-9：codes\13\13.3\ek-service-provider\src\main\java\org\crazyit\cloud\FirstServiceProvider.java

```
@SpringBootApplication
@EnableEurekaClient
@RestController
public class FirstServiceProvider {

    public static void main(String[] args) {
        // 读取控制台输入作为端口参数
        Scanner scan = new Scanner(System.in);
        String port = scan.nextLine();
        // 设置启动的服务器端口
        new SpringApplicationBuilder(FirstServiceProvider.class).properties(
                "server.port=" + port).run(args);
    }

    @GetMapping(value = "/person/{personId}", produces = MediaType.APPLICATION_
      JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId,
            HttpServletRequest request) {
        Person person = new Person(personId, "Crazyit", 30);
        person.setMessage(request.getRequestURL().toString());
        return person;
    }
}
```

由于服务提供者需要启动多个实例，因此我们修改了 main 方法，在启动时需要在控制台输入启动的端口。另外，修改了提供的服务，让 Person 对象保存 HttpServletRequest 的请求 URI。完成修改后，按照以下顺序启动服务：

1. 启动 Eureka 服务器（8761 端口）。
2. 启动第一个服务提供者，启动时在控制台输入：8081。
3. 启动第二个服务提供者，启动时输入 8082。
4. 启动服务调用者（9000 端口）。

成功启动后，在浏览器中访问：http://localhost:9000/router，可以看到如下输出：

```
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
```

进行多次刷新，可以看到端口会在 8081 与 8082 之间切换，说明我们的 Ribbon 已经生效。再次重申，Ribbon 的功能是从一堆服务器中选取一个服务器，至于选取出来的服务器做什么，Ribbon 并不关心。

13.4 客户端 Feign

在 Spring Cloud 集群中，各个角色之间的通信基于 REST 服务，因此在调用服务时，就不可避免地会使用 REST 服务的请求客户端。我们在前面的章节中使用了 Spring 自带的 RestTemplate，RestTemplate 使用 HttpClient 发送请求。本节将介绍另一个 REST 客户端：Feign。

13.4.1 Feign 介绍

Feign 是 Github 上的一个开源项目，该项目的目的是简化 Web Service 客户端的开发。在使用 Feign 时，可以使用注解来修饰接口，被注解修饰的接口具有访问 Web Service 的能力，这些注解既包括 Feign 自带的注解，也包括第三方的注解。除此之外，Feign 还支持插件式的编码器和解码器，我们可以通过该特性，对请求和响应进行不同的封装与解析。

Spring Cloud 将 Feign 集成到了 Netflix 项目中，如果与 Eureka、Ribbon 集成，Feign 就具有负载均衡的功能。Feign 本身在使用上的简便性，再加上与 Spring Cloud 的高度整合，大大降低了使用该框架在 Spring Cloud 中调用集群服务的开发工作量。

再次强调，Feign 只是一个 REST 客户端，主要用于调用 REST 服务。

13.4.2 第一个 Feign 程序

如前所述，Feign 是一个 REST 客户端，那么，我们先编写一个 REST 服务。本例的客户端将会访问 8080 端口的“/person/{personId}”服务。新建名称为“rest-server”的项目并添加服务，控制器与启动类内容请见代码清单 13-10。

代码清单 13-10：codes\13\13.4\rest-server\src\main\java\org\crazyit\cloud\ServerApp.java

```
@SpringBootApplication
@RestController
public class ServerApp {

    public static void main(String[] args) {
        new SpringApplicationBuilder(ServerApp.class).run(args);
    }
```

```
    @GetMapping(value = "/person/{personId}", produces = MediaType.APPLICATION_
JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId,
            HttpServletRequest request) {
        Person p = new Person(personId, "crazyit", 30);
        return p;
    }
}
```

启动 ServerApp，在浏览器中访问：http://localhost:8080/person/1，则可以看到输出的 JSON。接下来，建立第一个 Feign 程序，用来请求这个 REST 服务。新建 Maven 项目，加入以下依赖：

```
        <dependency>
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-core</artifactId>
            <version>9.5.1</version>
        </dependency>
        <dependency>
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-gson</artifactId>
            <version>9.5.1</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.16.18</version>
        </dependency>
```

新建服务请求客户端接口，请见代码清单 13-11。

代码清单 13-11：codes\13\13.4\feign-client\src\main\java\org\crazyit\cloud\PersonClient.java

```
/**
 * Person 客户端服务类
 * @author 杨恩雄
 */
public interface PersonClient {

    @RequestLine("GET /person/{personId}")
    Person findById(@Param("personId") Integer personId);

    // 为所有属性添加 setter 和 getter 等方法
    @Data
    class Person {
        Integer id;
        String name;
        Integer age;
    }
}
```

PersonClient 表示一个服务接口，在接口的“findById”方法中，使用了@RequestLine 注解，指示使用 GET 方法，向“/person/{personId}”发送请求，返回的是一个接口定义的 Person 类。这里为了减少代码量，使用了 Lombok 项目的@Data 注解修饰 Person。编写测试类，调用 PersonClient 请求服务，请见代码清单 13-12。

代码清单 13-12：codes\13\13.4\feign-client\src\main\java\org\crazyit\cloud\PersonMain.java

```
public class PersonMain {

    public static void main(String[] args) {
        PersonClient personClient = Feign.builder().decoder(new GsonDecoder())
                .target(PersonClient.class, "http://localhost:8080/");
        // 调用/person/{personId}服务
        Person person = personClient.findById(2);
        System.out
                .println(person.id + "---" + person.name + "---" + person.age);
    }
}
```

该测试类使用 Feign 的 API 创建一个 PersonClient 接口，再调用接口的 findById 方法请求服务，最后输出请求结果。再次强调，Feign 与 CXF 等框架类似，都是一个 REST 客户端，用来请求 REST 服务。但除此之外，它还可以与 Ribbon 一起使用，以请求微服务，我们下面讲解如何调用微服务。

➤➤ 13.4.3 使用 Feign 调用微服务

Feign 主要在服务调用者一端使用，在前面的例子中，都是使用 RestTemplate 调用微服务，而我们可以使用 Feign 来取代 RestTemplate。新建 Eureka 集群的各个角色。

- **Eureka 服务器**：端口为 8761，代码目录为：codes\13\13.4\ek-server。
- **服务提供者**：端口为 8080，代码目录为：codes\13\13.4\ek-service-provider。
- **服务调用者**：端口为 9000，通过浏览器访问，代码目录为：codes\13\13.4\ek-service-invoker。

集群各个角色的创建方法，请参考前面章节的介绍。服务提供者主要提供一个“/person/{personId}”的服务，在服务调用者里面，只需要加入 Feign 的 API 即可，Spring Cloud 已经对 Feign 进行了封装。服务调用者的 pom.xml 内容如下：

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
```

为服务调用者编写调用接口，请见代码清单 13-13。

代码清单 13-13：codes\13\13.4\ek-service-invoker\src\main\java\org\crazyit\cloud\PersonClient.java

```
//声明调用的服务名称
@FeignClient("service-provider")
public interface PersonClient {

    @RequestMapping(method = RequestMethod.GET, value = "/person/{personId}")
    Person findById(@PathVariable("personId") Integer personId);
}
```

与前面定义的接口不同，这里使用@FeignClient 修饰接口，并且定义了需要调用的服务。在接口方法中，使用@RequestMapping 与@PathVariable 来修饰方法与参数。可能大家对它们有点眼熟，这不是 Spring MVC 的注解吗？是的，Spring 为了让我们使用方便，实现了 Feign 的注解翻译器，使我们可以在接口中使用 Spring MVC 的注解，例如@RequestParam、@RequestHeader。

接下来，为服务调用者编写启动类与控制器，请见代码清单 13-14。

代码清单 13-14：codes\13\13.4\ek-service-invoker\src\main\java\org\crazyit\cloud\FirstInvoker.java

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
@EnableFeignClients
public class FirstInvoker {

    public static void main(String[] args) {
        SpringApplication.run(FirstInvoker.class, args);
    }

    @Autowired
    PersonClient personClient;

    @GetMapping(value = "/router", produces = MediaType.APPLICATION_JSON_VALUE)
    public Person router() {
        return personClient.findById(1);
    }
}
```

在启动类里面，使用了 EnableFeignClients 注解，表示开启 Feign 客户端配置，Spring 会自动帮我们将 Feign 配置好。在控制器或者其他的 bean 里面，直接注入 PersonClient 接口。本例中的 router 方法，用来调用 PersonClient 的 findById 方法。按照以下顺序测试本例。

① 启动 Eureka 服务器（8761 端口），代码目录为：codes\13\13.4\ek-server\src\main\java\org\crazyit\cloud\FirstServer.java。

② 启动服务提供者（8080 端口），代码目录为：codes\13\13.4\ek-service-provider\src\main\java\org\crazyit\cloud\FirstServiceProvider.java。

③ 启动服务调用者（9000 端口），代码目录为：codes\13\13.4\ek-service-invoker\src\main\java\org\crazyit\cloud\FirstInvoker.java

成功启动后，在浏览器中访问：http://localhost:9000/router，输出结果如下：

```
{"id":1,"name":"Crazyit","age":30}
```

实际上，Feign 只是取代了 RestTemplate 的角色，再次强调，它只是一个 REST 客户端，但经过加工后，它拥有访问微服务的能力。

13.5 容错框架 Hystrix

在很多的系统架构中，都需要考虑横向扩展、单点故障等问题。对于一个庞大的应用集群来说，部分服务或者机器出现问题不可避免，在出现故障时，如何减小故障的影响，保障集群的高可用，成为一个重要的课题。在微服务集群中，不管是服务器，还是客户端，都支持集群部署。本节将讲述 Spring Cloud 中的集群保护框架：Hystrix。

13.5.1 Hystrix 介绍

假设有一个应用程序，其中的调用关系如图 13-7 所示。

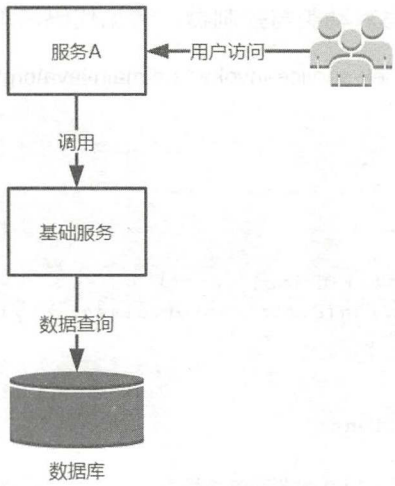


图 13-7 应用程序中的调用关系

在图 13-7 中，用户访问服务 A，服务通过 Web 接口或者其他方式访问基础服务模块，基础服务模块访问数据库。

如果数据库由于某些原因变得不可用，基础服务将会得到“数据库无法访问”的信息，并且会将此信息告知服务 A 模块。在出现问题时，用户不断地请求服务 A 模块，而服务 A 模块则继续请求基础服务模块，基础服务模块仍然不停地连接有问题的数据库直到超时。大量的用户请求（包括重试的请求）发送过来，使整个应用不堪重负。

实际情况可能更加糟糕，用户的请求不停地发送给服务 A 模块，而由于数据库的原因，基础服务模块迟迟无法响应，这有可能造成整个机房的网络阻塞，受害的不仅仅是应用程序，机房中的所有服务都有可能因为网络原因而瘫痪。我们需要寻找更好的方式解决这些问题。

Hystrix 是 Netflix 下的一个 Java 库，Spring Cloud 将 Hystrix 整合到 Netflix 项目中，Hystrix 通过添加延迟阈值以及容错的逻辑，来帮助我们控制分布式系统间组件的交互。Hystrix 通过隔离服务间的访问点，停止它们之间的级联故障，允许可回退操作来提供容错机制。使用 Hystrix 之后，程序结构如图 13-8 所示。

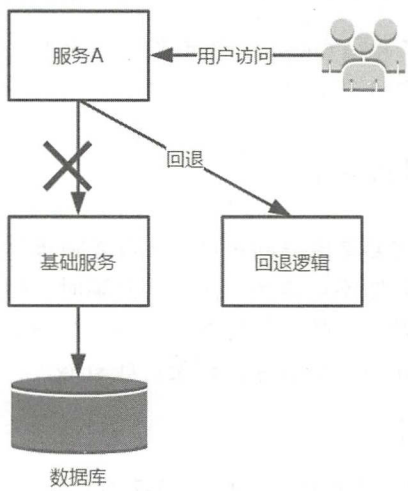


图 13-8 使用 Hystrix 的程序结构

在如图 13-8 所示的程序结构中，当基础服务模块或者数据库不可用时，服务 A 将对其进行“熔断”，这样在一定的时间内，服务 A 都不会再调用基础服务，以维持本身的稳定。

Hystrix 会在“服务调用者”一端使用，即图中的服务 A，如果服务 A 调用基础服务超时，则会进入回退逻辑。我们下面编写一个简单的 Hystrix 程序。

➤➤ 13.5.2 第一个 Hystrix 程序

新建一个普通的 Maven 项目，加入以下依赖：

```
        <dependency>
            <groupId>com.netflix.hystrix</groupId>
            <artifactId>hystrix-core</artifactId>
            <version>1.5.12</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <version>1.7.25</version>
            <artifactId>slf4j-log4j12</artifactId>
        </dependency>
```

编写一个 Hystrix 命令，用于模拟调用第三方服务，请见代码清单 13-15。

代码清单 13-15：codes\13\13.5\first-hystrix\src\main\java\org\crazyit\cloud\HelloCommand.java

```
/**
 * 定义一个 Hello 命令
 * @author 杨恩雄
 *
 */
public class HelloCommand extends HystrixCommand<String> {

    public HelloCommand() {
        // 调用父类的构造器，设置命令组的 key
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
    }

    protected String run() throws Exception {
        // 模拟调用第三方服务，调用超时
        Thread.sleep(5000);
        return "success";
    }

    protected String getFallback() {
        System.out.println("调用超时，执行 HelloCommand 的回退方法");
        return "error";
    }
}
```

在 HelloCommand 的 run 方法中，会设置线程睡眠 5s，最后返回“success”字符串。getFallback 方法是一个回退方法。当 run 方法调用超时时，将会执行 getFallback 方法的逻辑，在默认情况下，如果 run 方法在 1s 内没有执行完成，则会执行回退。接下来编写 main 方法来执行命令，请见代码清单 13-16。

代码清单 13-16：codes\13\13.5\first-hystrix\src\main\java\org\crazyit\cloud\HelloMain.java

```
public class HelloMain {
```

```
    public static void main(String[] args) {
        // 创建命令
        HelloCommand command = new HelloCommand();
        String result = command.execute();
        System.out.println("请求正常的服务，结果：" + result);
    }
}
```

运行代码清单 13-16，输出结果如下：

```
调用超时，执行 HelloCommand 的回退方法
请求正常的服务，结果：error
```

最终得到的结果为 error，而并非 run 方法返回的 success，由此可见执行了回退方法。如果我们将 HelloCommand 中的 run 方法的睡眠时间改为 500ms，再运行代码清单 13-16，可返回 success。

可以使用 Hystrix 改造前一小节的程序，将服务 A 调用基础服务的逻辑放到 run 方法中，这样当基础服务宕机时，就可以直接调用回退方法。

➤➤ 13.5.3 在 Spring Cloud 中整合 Hystrix

我们改造一下前面的项目，将 Hystrix 放入到 Spring Cloud 集群中。如前所述，Hystrix 在调用服务的一端使用，因此我们要在服务调用者里面加入 Hystrix。本例集群的几个项目描述如下。

- **Eureka 服务器**：8761 端口，代码目录为：codes\13\13.5\hy-server。
- **服务提供者**：8080 端口，代码目录为：codes\13\13.5\hy-service-provider。它提供一个“/person/{personId}”的服务，返回 Person 的 JSON 字符串。
- **服务调用者**：9000 端口，代码目录为：codes\13\13.5\hy-service-invoker。它提供一个“/router”的服务，该服务可通过浏览器访问。

集群中的这三个角色，与本章前面项目中的类似。我们要修改服务调用者，为其加入 Hystrix，本例服务调用的 pom.xml 依赖如下：

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
        </dependency>
        <dependency>
```

```
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        </dependency>
    </dependencies>
```

粗体字代码为加入的 Hystrix 依赖，修改 application.yml，内容请见代码清单 13-17。

代码清单 13-17：codes\13\13.5\hy-service-invoker\src\main\resources\application.yml

```
server:
  port: 9000
spring:
  application:
    name: hy-service-invoker
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
feign:
  hystrix:
    enabled: true
```

由于服务调用者使用 Feign 调用服务，因而整合 Hystrix 后要打开开关，以上的粗体字配置，即是用来打开 Feign 的 Hystrix 开关的。接下来，为启动类加入 Hystrix 注解，请见代码清单 13-18。

代码清单 13-18：codes\13\13.5\hy-service-invoker\src\main\java\org\crazyit\cloud\FirstInvoker.java

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
@EnableFeignClients
@EnableCircuitBreaker
public class FirstInvoker {

    public static void main(String[] args) {
        SpringApplication.run(FirstInvoker.class, args);
    }

    @Autowired
    PersonClient personClient;

    @GetMapping(value = "/router", produces = MediaType.APPLICATION_JSON_VALUE)
    public Person router() {
        return personClient.findById(1);
    }
}
```

启动类与前面的例子类似，这里仅仅加入了一个 EnableCircuitBreaker 的注解，表示开启 Hystrix 的断路器。下面修改 Feign 接口，为其加入回退逻辑，请见代码清单 13-19。

代码清单 13-19：codes\13\13.5\hy-service-invoker\src\main\java\org\crazyit\cloud\PersonClient.java

```
@FeignClient(name = "hy-service-provider", fallback = PersonClientFallback.class)
public interface PersonClient {

    @RequestMapping(method = RequestMethod.GET, value = "/person/{personId}")
    Person findById(@PathVariable("personId") Integer personId);

    @Component
    static class PersonClientFallback implements PersonClient {

        @Override
        public Person findById(Integer personId) {
            // 回退方法
            System.out.println("执行 findByid 的回退方法，返回一个预先设定好的 Person");
            Person p = new Person(-1, "模拟 Person", 0);
            return p;
        }
    }
}
```

在代码清单 13-19 中新建了一个 PersonClientFallback 类，该类实现了 PersonClient 接口，当调用 findById 方法（即调用“/person/{personId}”服务）超时时，就会调用 PersonClientFallback 的 findById 方法，返回一个预先设定好的 Person 对象，也就是说，PersonClientFallback 类是一个备胎。按照以下的步骤测试 Hystrix。

① 启动 Eureak 服务器，8761 端口，代码目录为：codes\13\13.5\hy-server\src\main\java\org\crazyit\cloud\FirstServer.java。

② 启动服务调用者，9000 端口，代码目录为：codes\13\13.5\hy-service-invoker\src\main\java\org\crazyit\cloud\FirstInvoker.java。

这里的目的是测试“服务调用者”的回退逻辑是否被触发，因此不需要启动服务提供者。成功启动上面两个程序后，在浏览器中访问：http://localhost:9000/router，浏览器输出如下：

```
{"id":-1,"name":"模拟 Person","age":0}
```

根据结果可知，备胎 PersonClientFallback 类的 findById 方法被调用了。我们可以启动服务提供者，代码目录为：codes\13\13.5\hy-service-provider\src\main\java\org\crazyit\cloud\FirstServiceProvider.java。使用浏览器再次访问 9000 端口的 router 服务，可以看到回退逻辑已经不再执行。

注意：如果后面再次启动服务提供者，由于缓存原因，需要多刷新几次才能看到结果。

13.6 本章小结

本章讲解了如何使用 Spring Cloud 开发微服务，主要讲解了 Eureka、Ribbon、Feign 与 Hystrix 这几个框架。我们需要大概了解这 4 个框架的基本作用，你可以仿照本章的例子，使用各个框架来发布、调用微服务。

Spring Cloud 是微服务开发的工具箱，它不止有这 4 个基本框架，还有很多的框架，例如，Sleuth、Spring Cloud Config、网关等。关于 Spring Cloud 的更多内容，可以参考笔者的另外一本书：《疯狂 Spring Cloud 微服务架构实战》。

CHAPTER

14

第14章 实战案例

本章要点

- 系统功能及界面
- 数据库设计
- 图书管理功能
- 入库与销售功能

学习了前面章节的知识后，这一章我们以一个图书进销存系统完成本书的讲解。该系统会综合运用前面介绍的 Spring Boot、Thymeleaf 与 JavaScript 等框架。本案例所使用的技术及相应的版本如下。

- Spring Boot 2.0：笔者成书时最新的 Spring Boot 版本，它会自动引入 Spring 5.0、Spring MVC 5.0 等组件。
- Spring Data 2.0：使用“spring-boot-starter-data-jpa”依赖，自动引入 Spring Data 2.0。
- Thymeleaf 3.0：使用“spring-boot-starter-thymeleaf”依赖，自动引入 Thymeleaf 3.0。
- MySQL 5.6
- jQuery 3.3.1：本书所使用的 jQuery 版本。
- Bootstrap 4.0.0：本书前面章节所使用的 Bootstrap 版本。

为简单起见，本案例并没有整合 Spring Cloud，只是一个单体应用。读者学习完本案例后，可以结合 Spring Cloud 来改进这个图书进销存系统，建议参考笔者的另一本书：《疯狂 Spring Cloud 微服务架构实战》进行改进。

14.1 系统功能及界面

图书进销存系统主要包括图书管理、入库与销售模块，在实际应用中，一个系统所含有的业务模块可能更为复杂，但万变不离其宗，大家可根据本章所掌握的知识，开发其他更加复杂的功能。接下来，我们介绍本章案例的功能及原型。

14.1.1 登录与首页

本案例的原型目录为：codes\14\book-prototype，读者可直接到该目录下获取静态的 HTML 页面。登录页面使用 Bootstrap 的表单组件，界面请见图 14-1。



图 14-1　登录界面

登录后，进入系统主界面，主界面请见图 14-2。



图 14-2 主界面

主界面的最上面是 banner 与导航，在这里会显示当前部分图书的库存信息，右边显示当前最新添加的图书。该系统使用了 Bootstrap 的轮播插件，该轮播插件内容如代码清单 14-1 所示。

代码清单 14-1：codes\14\book-prototype\templates\main.html

```
    <div id="carouselExampleIndicators" class="carousel slide" data-interval="5000">
        <ol class="carousel-indicators">
            <li  data-target="#carouselExampleIndicators"  data-slide-to="0"  class=
"active"></li>
            <li data-target="#carouselExampleIndicators" data-slide-to="1"></li>
            <li data-target="#carouselExampleIndicators" data-slide-to="2"></li>
        </ol>
        <div class="carousel-inner" style="margin-left: 60px;">
            <div class="carousel-item active">
                <img class="d-block" width="500" src="../static/images/slide1.jpg">
            </div>
            <div class="carousel-item">
                <img class="d-block" width="500" src="../static/images/slide2.jpg">
            </div>
            <div class="carousel-item">
                <img class="d-block" width="500" src="../static/images/slide3.jpg">
            </div>
        </div>
        <a class="carousel-control-prev" href="#carouselExampleIndicators" role="button"
data-slide="prev">
            <img src="../static/images/back.png"/>
        </a>
        <a class="carousel-control-next" href="#carouselExampleIndicators" role=
"button" data-slide="next">
            <img src="../static/images/next.png"/>
        </a>
    </div>
```

轮播插件中的 ol 节点是轮播内容的控制器，div 节点中为轮播的图片内容，两个 a 节点为向左与向右的控制按钮。

14.1.2 图书管理模块

图书管理模块主要有数据列表、新建、修改、查看与删除功能，其中修改与查看功能可以共用新建界面，因此只提供列表与新建界面，如图 14-3 与图 14-4 所示。



图 14-3　图书列表界面



图 14-4　新建图书界面

图书列表与新建图书界面，使用了 Bootstrap 的列表与表单。关于这两个界面的实现，请参见本书的第 11 章内容，在此不贴出界面代码了。

14.1.3　入库模块

入库模块主要有列表与新建入库单功能，它们的界面如图 14-5 与图 14-6 所示。



图 14-5　入库单列表界面



图 14-6 新建入库单界面

在入库单列表界面，会显示该入库单所有的图书，并会显示该入库单所有图书的入库总数量。在新建入库单界面，选择入库图书，单击“选择图书”按钮后，会弹出图书选择界面。

14.1.4 销售模块

销售模块与入库模块类似，在一次图书销售的过程中，同样可以选择多本图书，其列表与新建功能的界面如图 14-7 与图 14-8 所示。



图 14-7 销售列表



图 14-8 新建销售

14.1.5 导航组件与分页组件

前面介绍的几个模块，都使用了 Bootstrap 的导航组件与分页组件。在这里介绍一下这两个组件的代码，以下为导航组件的代码：

```
    <nav class="navbar navbar-expand-lg navbar-light" style="background-color:
#A9C8E7;">
        <div class="navbar-nav">
            <a class="nav-item nav-link active" href="#">首页</a>
            <a class="nav-item nav-link" href="book/list.html">图书管理</a>
            <a class="nav-item nav-link" href="store/list.html">图书入库</a>
            <a class="nav-item nav-link" href="sale/list.html">图书销售</a>
        </div>
    </nav>
```

在导航组件中，如果要设置某个菜单为选中状态，则可以为其添加“active”的 class 属性。分页组件代码如下：

```
    <nav class="col-lg-1">
        <ul class="pagination">
            <li class="page-item">
                <a class="page-link" href="#" aria-label="Previous">
                    <span aria-hidden="true">&laquo;</span>
                </a>
            </li>
            <li class="page-item active "><a class="page-link" href="#">1</a></li>
            <li class="page-item"><a class="page-link" href="#">2</a></li>
            <li class="page-item"><a class="page-link" href="#">3</a></li>
            <li class="page-item"><a class="page-link" href="#">4</a></li>
            <li class="page-item">
                <a class="page-link" href="#" aria-label="Next">
                    <span aria-hidden="true">&raquo;</span>
                </a>
            </li>
        </ul>
    </nav>
```

分页组件与导航组件类似，如果想设置某一页为选中状态，则可以为 li 节点添加“active”的 class 属性。

这个图书进销存系统主要有图书管理、入库与销售这三个模块，大家知道这几个模块的功能即可。这个系统并没有相应的需求文档，我们可直接按照原型界面来设计数据库，并按其来实现功能。

14.1.6 运行本章案例

如果你想查看最终的项目效果，则可以按照以下步骤操作：

① 在 MySQL 数据库中新建一个 BMS 库，案例默认的数据库访问用户为 root，密码为 123456。

② 将 codes\14\BMS.sql 导入 BMS 库。

③ 如果你的数据库用户名与密码不是 root/123456，则修改 codes\14\book.jar 里面的配置文件，目录为：book.jar\BOOT-INF\classes\application.yml，将 spring.datasource.username 与 password 改为你的用户名与密码，将修改后的 application.yml 文件拉回 jar 包中。

④ 使用命令行工具，进入 book.jar 所在目录，执行命令：java –jar book.jar，成功运行后，使用地址 http://localhost:8080/访问系统。

⑤ 登录的用户名为 admin，密码为 123456。

另外，也可以将 codes\14\book 项目导入 Eclipse 中。运行 codes\14\book\src\main\java\org\crazyit\book\BookApp.java 类的 main 方法，同样可以启动系统。

14.2 数据库设计

主要有以下几个数据表：用户表、图书表、库存表、入库单与销售单，我们先简单介绍这些数据表的结构。

注：本章案例的 SQL 脚本存放在 codes\14 目录下，名称为 BMS.sql，你可以直接将该脚本导入 MySQL 数据库。

14.2.1 用户表

用户表用来保存系统的用户信息，为了简单起见，本例的 USER 表只包含三个字段。

- ID：主键，本系统中所有数据表的 ID 字段都是 int 类型，并且是自增长的。
- NAME：用户名。
- PASSWD：密码。

在本系统中，用户表主要用来实现登录功能，用于验证用户名和密码，并没有其他的业务含义，读者可以根据自己系统的实际情况来设计用户表。

14.2.2 图书表与库存表

图书表用于保存图书的基本数据，本例的 BOOK 表有以下字段。

- ID：主键。

- NAME：图书名称。
- COVER：图书封面图片，该字段只保存图片的 URL，并不保存图片内容。
- AUTHOR：图书作者。
- PRICE：图书价格。
- IS_DELETE：是否删除该图书数据，0 为 false，1 为 true。

还需要添加一个库存表，用于保存图书的库存信息，库存表名称为 BOOK_STOCK，有以下字段。

- ID：主键。
- BOOK_ID：图书的 ID，表示该库存对应的图书记录。
- STOCK_NUM：图书的库存数量。

在实际应用中，有一些业务系统，可能需要记录每一笔数据的进销存明细，这样的话，我们还要建立相应的进销存流水表，但本案例并不涉及该功能。

14.2.3 入库单数据表

一张入库单可录入多本图书，因此我们需要建立一个主表和一个明细表来保存入库单数据与明细数据。主表 STORE 有如下字段。

- ID：主键。
- CREATE_TIME：创建时间。

本例中的入库单只有两个字段，而在实际应用中，可能还需要增加入库单编码、创建人等字段，这里我们为了简单起见，省略了这些字段。入库单明细表 STORE_ITEM 包含如下字段。

- ID：主键。
- BOOK_ID：入库单明细中的关联图书数据 ID。
- STORE_ID：明细数据所属的入库单。
- AMOUNT：该条明细数据对应书本的入库数量。

在入库单的列表界面中，需要显示该入库单的全部入库数量，而在查询时，需要计算该入库单全部明细的图书数量，我们在后面实现功能时要考虑这一点。

14.2.4 销售单数据表

销售单与入库单类似，也有一个主表和一个明细表，销售单表 SALE 包含的字段如下。

- ID：主键。
- CREATE_TIME：创建时间。

明细表 SALE_ITEM 包含的字段如下。

- ID：主键。
- BOOK_ID：明细数据对应的图书 ID。
- SALE_ID：该明细数据所属的销售单。
- AMOUNT：该图书的销售数量。



仅供非商业用途或交流学习使用

➢ PRICE：在销售时该图书的价格。

至此，数据库表已经介绍完了，它们的结构都比较简单，你可以直接将 BMS.sql 导入 MySQL 中。下面开始构建这个项目。

14.3 搭建项目框架

首先搭建项目框架，实现最简单的用户登录功能。

14.3.1 项目依赖与项目结构

新建名称为“book”的 Maven 项目，本例会使用 Thymeleaf、Spring Data 等框架。本例的 pom.xml 文件的内容请见代码清单 14-2。

代码清单 14-2：codes\14\book\pom.xml

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.1.RELEASE</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- Thymeleaf -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <!-- Spring Data JPA -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <!-- MySQL 驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>
        <!-- 文件上传组件 -->
        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
            <version>1.3.3</version>
        </dependency>
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
        </dependency>
</dependencies>
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
<build>
    <finalName>book</finalName>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Spring Boot 会帮我们自动引入大部分的依赖，包括 Thymeleaf、Spring Data、Spring MVC 等。接下来，为项目加入各个目录，项目的目录结构如图 14-9 所示。

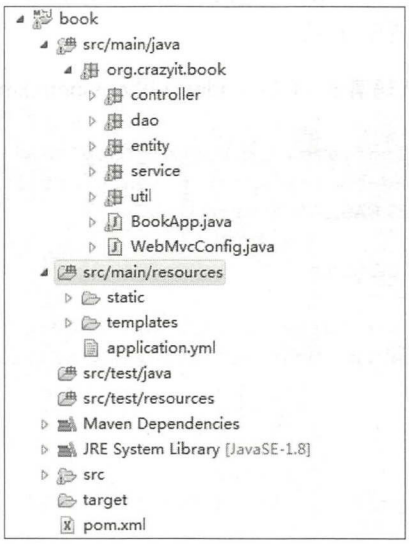


图 14-9 项目目录结构

项目的 src/main/java 目录下有 controller、dao、entity、service 等目录，分别用于存放控制器、DAO 对象、映射实体与服务类，这与我们传统的 Java EE 应用结构类似。

图书进销存系统的启动类为 codes\14\book\src\main\java\org\crazyit\book\BookApp.java。在打包之前，直接在 Eclipse 中运行它的 main 方法，并在浏览器中打开：http://localhost:8080，即可访问进销存系统的登录界面。

14.3.2 项目配置

项目的配置文件为 src/main/resources/application.yml，其主要用于配置数据库的相关信息。配置文件内容请见代码清单 14-3。

代码清单 14-3：codes\14\book\src\main\resources\application.yml

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/BMS
    username: root
    password: 123456
  jpa:
    show-sql: false
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

配置项前缀 sprjng.datasource.*，用于配置数据源。各个配置项，看其名字就能明白其含义。代码清单 14-3 主要用来配置驱动类、数据库 URL、用户名与密码。最下面有一个 spring.jpa.show-sql 项，用于配置是否在控制台中输出执行的 SQL 语句，该项主要用于调试。

14.3.3 实体设计

前面设计好了数据表，现在需要建立与数据表对应的实体类。每个表都有一个 ID 字段。我们可以新建一个基类，请见代码清单 14-4。

代码清单 14-4：codes\14\book\src\main\java\org\crazyit\book\entity\BaseEntity.java

```
/**
 * 该类是所有实体类的父类，它不会映射到表，但其属性会映射到子类的相应字段
 * @author 杨恩雄
 */
@MappedSuperclass
public class BaseEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    ...省略 setter 与 getter 方法
}
```

BaseEntity 类使用@MappedSuperclass 修饰，表示它是其他实体类的父类，它本身并不会映射到数据表，其他实体类直接继承 BaseEntity。以 Book 表的映射类 Book 为例，按相应的字段建立属性即可，请见代码清单 14-5。

代码清单 14-5：codes\14\book\src\main\java\org\crazyit\book\entity\Book.java

```
/**
 * Book 表
 * @author 杨恩雄
 */
@Entity
@Table(name = "BOOK")
public class Book extends BaseEntity {

    @Column
    private String name;

    @Column
    private String cover;

    @Column
    private String author;

    @Column
    private BigDecimal price;

    @Column
    private Boolean isDelete = false;

    ...省略 setter 与 getter 方法
}
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

按照同样的方式，建立入库单与入库单明细两个实体，请见代码清单 14-6。

代码清单 14-6：codes\14\book\src\main\java\org\crazyit\book\entity\Store.java

```
/**
 * 入库单
 * @author 杨恩雄
 */
@Entity
@Table(name = "STORE")
public class Store extends BaseEntity {

    @Column
    private Date createTime;

    // 非持久化字段，多本书之间用逗号隔开
    @Transient
    private String bookNames;

    // 非持久化字段，保存入库单总数量
    @Transient
    private int amount;

    ...省略 setter 与 getter 方法
}
```

codes\14\book\src\main\java\org\crazyit\book\entity\StoreItem.java

```
/**
 * 入库单明细
 * @author 杨恩雄
 */
@Entity
@Table(name = "STORE_ITEM")
public class StoreItem extends BaseEntity {

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "BOOK_ID")
    private Book book;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "STORE_ID")
    private Store store;

    @Column
    private Integer amount;

    ...省略 setter 与 getter 方法
}
```

Store 类中有两个字段：bookNames 与 amount，bookNames 用于在界面中显示该入库单中所有明细的图书名称，amount 则表示整张入库单的入库数量。StoreItem 会与 Book 和 Store 两个实体关联，此处的关系是单向一对多关系。

按照同样的方式，建立销售单相关的表：SALE 与 SALE_ITEM，包含的字段与入库单两个表类似，在此不再赘述。除此之外，还要建立库存实体、用户实体，对应的 Java 类分别为 BookStock 与 User。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

14.3.4 用户登录

我们实现一个最简单的用户登录功能。根据界面中输入的用户名和密码，到数据库中查询相应的用户数据，若查询到，则跳转到首页。DAO 与 Service 代码请见代码清单 14-7。

代码清单 14-7：codes\14\book\src\main\java\org\crazyit\book\dao\UserDao.java

```
public interface UserDao extends JpaRepository<User, Integer> {

    /**
     * 根据用户名和密码查询用户数据
     */
    public User findByNameAndPasswd(String name, String passwd);
}
```

codes\14\book\src\main\java\org\crazyit\book\service\UserService.java

```
@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    public User login(User user) {
        User u = userDao.findByNameAndPasswd(user.getName(), user.getPasswd());
        return u;
    }
}
```

在控制器中，调用 UserService 的 login 方法，如果该方法返回 null，则表示没有查询到相应用户，则登录失败，反之则表示登录成功，请见代码清单 14-8。

代码清单 14-8：codes\14\book\src\main\java\org\crazyit\book\controller\UserController.java

```
@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/")
    public String index(@ModelAttribute User user) {
        return "index";
    }

    @PostMapping(value = "/login")
    public String login(@ModelAttribute User user, HttpSession session, Model model) {
        User loginUser = userService.login(user);
        if(loginUser == null) {
            // 登录失败，返回登录界面
            model.addAttribute("msg", "用户名密码错误");
            return "index";
        } else {
            // 登录成功
            session.setAttribute("user", loginUser);
            return "main";
        }
    }
}
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

登录页面为 index.html，表单的主要内容如下：

```
<form action="main.html" th:action="@{/login}" th:method="post" th:object= "${user}">
      <span class="col-sm-4" style="color: red;" th:text="${msg}">用户名密码错误</span>
      <input type="text" class="form-control" name="name" th:value="*{name}">
      <input type="password" class="form-control" name="passwd" th:value="*{passwd}">
      <button type="submit" class="btn btn-primary">登录</button>
</form>
```

为了减少篇幅，以上仅贴出 index.html 的部分内容。在笔者提供的数据脚本中，包含一个 admin 用户，密码为 123456，读者在使用系统时，可直接使用该用户登录。在实际应用中，可能还会加密密码。

14.3.5 片断定义

该系统中有许多共同的部分，例如主界面最上面的 banner、导航栏，可以将这些共同部分提取出来定义为单独的片断，而在每个页面里面，可以使用 Thymeleaf 的 th:replace 等属性来引用这些片断。将 banner 的内容定义为一个 HTML 页面，请见代码清单 14-9。

代码清单 14-9：codes\14\book\src\main\resources\templates\common\banner.html

```
<div class="row">
   <div class="col-lg-4" style="text-align: center;">
      <img th:src="@{/images/logo.png}"/>
   </div>
   <div class="col-lg-8" style="text-align: center;">
      <img th:src="@{/images/banner.png}"/>
   </div>
</div>
```

在具体的页面中，使用 th:replace 属性来引用片断：

```
<div class="row" th:replace="~{common/banner}">
   <div class="col-lg-4" style="text-align: center;">
      <img src="../static/images/logo.png"/>
   </div>
   <div class="col-lg-8" style="text-align: center;">
      <img src="../static/images/banner.png"/>
   </div>
</div>
```

这是 HTML 原型中的代码，仅加入了 th:replace 属性，这种方式并没有破坏原型的结构，依然可以在浏览器中直接打开该页面，并且效果与原型一致。

使用同样的方式，将导航栏抽取为一个独立的 HTML 页面，请见代码清单 14-10。

代码清单 14-10：codes\14\book\src\main\resources\templates\common\nav.html

```
<div class="row" th:fragment="nav(module)">
   <div class="col-lg-1"></div>
   <div class="col-lg-10">
      <nav class="navbar navbar-expand-lg navbar-light" style="background-color:
        #A9C8E7;">
          <div class="navbar-nav">
              <a id="main" class="nav-item nav-link" th:href="@{/main}">首页</a>
              <a id="book" class="nav-item nav-link" th:href="@{/book/list}">
               图书管理</a>
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
                <a id="store" class="nav-item nav-link" th:href="@{/store/list}">
                 图书入库</a>
                <a id="sale" class="nav-item nav-link" th:href="@{/sale/list}">
                 图书销售</a>
            </div>
        </nav>
    </div>
    <script>
        var module = '[[ ${module} ]]';
        $("#" + module).addClass("active");
    </script>
</div>
```

代码清单 14-10 定义了导航栏的片断，如果调用该片断，则需要传入相应的 module 名称，在片断中定义的 JavaScript，会根据传入的 module 名称，修改相应 a 节点的 class，让当前菜单变为选中状态。使用下面的代码调用该导航片断：

```
<div class="row" th:replace="~{common/nav::nav('main')}">
    <div class="col-lg-1"></div>
    <div class="col-lg-10">
        <nav class="navbar navbar-expand-lg navbar-light" style="background-color:
         #A9C8E7;">
            <div class="navbar-nav">
                <a class="nav-item nav-link active" href="#">首页</a>
                <a class="nav-item nav-link" href="book/list.html">图书管理</a>
                <a class="nav-item nav-link" href="store/list.html">图书入库</a>
                <a class="nav-item nav-link" href="sale/list.html">图书销售</a>
            </div>
        </nav>
    </div>
</div>
```

同样地，这个代码片断只是在原型的基础上，加入了 th:replace 属性，这并不会破坏原型的结构。当在浏览器中打开页面时，这里的 HTML 代码生效，经过 Thymeleaf 处理后，代码清单 14-10 生效。系统中的大部分页面都会使用 banner 与导航片断。

14.4 图书管理

这一节实现图书管理功能，在该模块中包含了列表、新建、图片上传等子功能。

14.4.1 数据列表

为了实现分页功能，需要对数据进行封装，Spring Data 已经为我们提供了数据封装的方法及对象，我们只需要按规范来封装数据，就可以直接在界面中实现分页查询。Service 方法请见代码清单 14-11。

代码清单 14-11：codes\14\book\src\main\java\org\crazyit\book\service\BookService.java

```
@Service
public class BookService {

    @Autowired
    private BookDao bookDao;
```


仅供非商业用途或交流学习使用

```
    public Page<Book> findAll(Pageable pageable) {
        Example<Book> example = getNotDeleteExample();
        return bookDao.findAll(example, pageable);
    }

    /**
     * 返回一个将 isDelete 设置为 false 的 Example
     */
    private Example<Book> getNotDeleteExample() {
        // 只查询没有删除的图书
        Book book = new Book();
        book.setIsDelete(false);
        return Example.of(book);
    }
}
```

在 Service 方法中，提供了一个 Example 对象，这是 QBE（Query By Example）的查询方式，其根据传入的样本对象进行查询，本例中的样本对象是一个 Book 实例。该实例将 isDelete 属性设置为 false，简单来说，就是查询 BOOK 表中 isDelete（IS_DELETE）字段为 false（0）的数据。

在调用 findAll 方法时，除需要传入样本对象外，还需要使用 Pageable 对象，该对象保存了分页的请求参数，例如当前的请求页、每页的数据量等。最后，findAll 方法会返回一个 Page 对象，该对象保存了分页的查询结果，例如查询的集合数据、总页数、总记录数等。注意，findAll 方法是 BookDao 的父接口 JpaRepository 提供的方法，并不需要额外定义。

接下来，在控制器中调用 Service 方法返回查询结果，请见代码清单 14-12。

代码清单 14-12：codes\14\book\src\main\java\org\crazyit\book\controller\BookController.java

```
@Controller
@RequestMapping("/book")
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping("/list")
    public String list(Model model,
            @PageableDefault(size = BookApp.PAGE_SIZE) Pageable pageable) {
        model.addAttribute("datas", bookService.findAll(pageable));
        return "book/list";
    }
}
```

在控制器的 list 方法中，首先为方法提供一个 Pageable 参数，使用 PageableDefault 注解来设置默认的分页参数，本例设置了每页数据量为 5。代码清单 14-13 在列表中遍历数据。

代码清单 14-13：codes\14\book\src\main\resources\templates\book\list.html

```
<tbody th:remove="all-but-first">
    <tr th:each="data : ${datas.content}" th:object="${data}">
        <td scope="row"><input type="checkbox" name="ids" th:value="*{id}"/></td>
        <td><a href="view.html" th:href="@{/book/view/} + *{id}"
        target="_blank" th:text="*{name}">疯狂 Spring Cloud 微服务架构实战</a></td>
        <td th:text="*{author}">杨恩雄</td>
        <td th:text="*{price}">40</td>
```


仅供非商业用途或交流学习使用

```
            <td><a th:href="@{/book/edit/} + *{id}">修改</a></td>
        </tr>
    </tbody>
```

查询的集合数据会被封装在 Page 对象的 content 属性中，因此要对 content 进行遍历。在 Thymeleaf 中遍历数据较为简单，直接输出相应的属性即可。注意，在 tbody 节点中加入 th:remove 属性，目的是为了保证该模板能同时提供给 UI 设计师和程序员使用，经过 Thymeleaf 处理后，会保留 tbody 下的第一个 tr，其余 tr 将会全部被删除。

启动服务器，在浏览器中访问：http://localhost:8080/book/list，即可看到效果。如果需要请求下一页，则访问：http://localhost:8080/book/list?page=1。

14.4.2 通用的分页组件

多个列表界面都需要实现分页功能，我们可以编写一个通用的分页组件，让每个列表界面直接引用该分页组件，这样可以减少许多重复的工作。对于每一个列表界面，不同的是请求数据的链接，因此我们可以把请求数据的 URL 作为参数。我们在代码清单 14-14 中定义了分页组件的模板。

代码清单 14-14：codes\14\book\src\main\resources\templates\common\page.html

```
<nav th:fragment="page(url)">
    <ul class="pagination">
    <li class="page-item">
        <a class="page-link" href="#" tabindex="-1"
        th:href="@{${url}} + |?page=| + ${datas.number < 1 ? 0 : (datas.number - 1)}">
         上一页</a>
    </li>
    <li class="page-item"
        th:each="pageNum : ${#numbers.sequence(0, datas.totalPages - 1)}"
        th:class="|page-item| + ${pageNum == datas.number ? ' active' : ''}">
        <a class="page-link" href="#" th:text="${pageNum + 1}" th:href="@{${url}}
         + |?page=| + ${pageNum}">1</a>
    </li>
    <li class="page-item">
        <a class="page-link" href="#"
        th:href="@{${url}} + |?page=| + ${datas.number >= (datas.totalPages - 1) ?
         datas.number : (datas.number + 1)}">下一页</a>
    </li>
    <li class="page-item">
        <a class="page-link" href="#" th:text="|共 | + ${datas.totalPages} + | 页
         |">共 10 页</a>
    </li>
    </ul>
</nav>
```

要使用分页组件的片断，需要传入列表的 URL，例如在前一小节中，查询图书的链接为“/book/list”，那么在调用片断时，只需要使用“th:replace="~{common/page::page('/book/list')}"”，就可以将图书的列表链接传到片断中。

在前一小节我们介绍过，如果要请求第二页，则需要这个地址：/book/list?page=1，而所有的分页结果，都保存在 Page 对象里面。分页组件会获取 Page 对象（datas）的部分属性来生成模块，例如，代码清单 14-14 中的所有 datas 变量均是 Page 对象。我们可以在页面中使用

仅供非商业用途或交流学习使用

Page 对象的以下属性。

- number：当前页，从 0 开始。
- totalPages：总页数。
- totalElements：总记录数。
- size：每页数据量。

本例中的图书列表使用该分页组件后，效果如图 14-10 所示。



图 14-10　分页效果

我们只要知道 Page 属性，就可以定义出各种交互模式的分页组件，读者可以在此基础上尝试开发其他的分页组件。

14.4.3　图片上传

在新建图书时，会涉及图片的上传功能，下面我们先谈一下图片上传的实现逻辑。在页面中，通过 file 控件选中图片后，再单击“上传”按钮可实现 AJAX 的图片上传。服务器在处理图片时，将图片通过 IO 流写入服务器的文件中，最后将图片的相对路径返回给页面。在这个过程中，要配置本地服务器的资源访问路径，例如把图片保存到 D 盘的 book 目录下。那么为了让这个目录下的所有文件可以通过 Web 的方式来访问，就需要添加映射关系，于是，新建一个配置类，请见代码清单 14-15。

代码清单 14-15：codes\14\book\src\main\java\org\crazyit\book\WebMvcConfig.java

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

    // 保存上传文件的目录
    public final static String FILE_DIR = "D:/book/upload/";

    /**
     * 配置静态资源的 Web 访问路径，例如将上传的文件 abc.png 保存在 D:/book/upload 下
     * 那么在浏览器中访问的路径为：http://localhost:8080/upload/abc.png
     */
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/upload/**").addResourceLocations(
                "file:///" + WebMvcConfig.FILE_DIR);
    }
}
```


仅供非商业用途或交流学习使用

在该配置文件中，配置了 D:/book/upload 目录，其对应的 Web 访问路径为“/upload/**”，例如将一个 abc.png 文件放到 D:/book/upload 目录下，那么启动服务器后，我们可以在浏览器中打开：http://localhost:8080/upload/abc.png 来访问这个图片文件。在处理上传文件时，我们只需要将文件保存到 D:/book/upload 目录下，然后将“/upload/文件名”返回给页面即可。

实现 AJAX 文件上传较为简单，其中页面与控制器代码请见代码清单 14-16。

代码清单 14-16：codes\14\book\src\main\resources\templates\book\content.html

```
<div class="col-sm-7">
    <input id="cover" th:disabled="${readonly}" type="file" class="form-control"/>
    <input type="hidden" value="" id="coverPath" name="cover" th:value="*{cover}"/>
</div>
<script>
    $("#uploadButton").click(function() {
        var formData = new FormData();
        var cover = $("#cover")[0].files[0];
        formData.append("cover", cover);
        $.ajax({
            url:'[[ @{/book/upload} ]]',
            data: formData,
            type: 'POST',
            contentType: false,
            processData: false,
            success: function(obj){
                $("#coverPath").val(obj);
                $("#coverImg").attr("src", obj);
            }
        });
    });
</script>
```

codes\14\book\src\main\java\org\crazyit\book\controller\BookController.java

```
    @RequestMapping("/upload")
    public void upload(@RequestParam("cover") MultipartFile uploadFile,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
        byte[] content = uploadFile.getBytes();
        // 保存文件到具体目录，此处为 D:/book/upload
        String path = WebMvcConfig.FILE_DIR;
        File folder = new File(path);
        if (!folder.exists()) {
            folder.mkdirs();
        }
        // 获取文件后缀
        String fileSuffix = FileUtil.getSuffix(uploadFile.getOriginalFilename());
        // 设置文件名
        File file = new File(folder.getAbsolutePath() + File.separator
                + UUID.randomUUID().toString() + fileSuffix);
        file.createNewFile();
        // 写到服务器文件
        FileUtil.writeFile(file, content);
        response.getWriter().write("/upload/" + file.getName());
    }
```

在页面中，单击“上传”按钮后，会发送 AJAX 请求到 upload 方法，upload 方法在处理

请求时，将上传的文件保存到本地服务器的目录中，本例的目录为“D:/book/upload”，最后将“/upload/文件名”字符串返回给页面。注意，为了避免文件名重复，每一个上传的文件，都会使用 UUID 作为文件名。启动服务器，在浏览器中访问：http://localhost:8080/book/add，单击“上传”按钮上传文件，结果如图 14-11 所示。

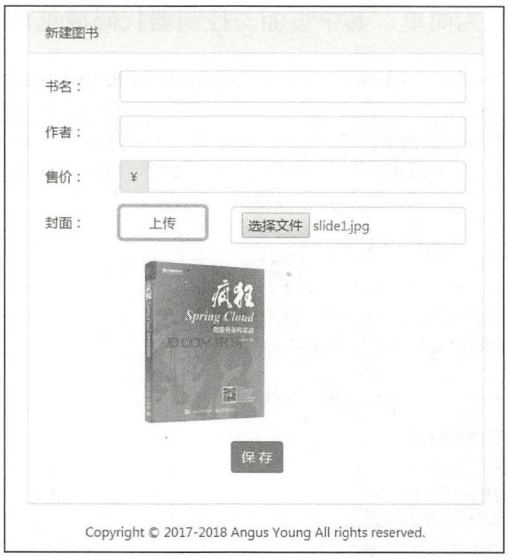


图 14-11　图片上传

在图片上传的 AJAX 回调方法中，需要注意一点，当上传成功后，会将“/upload/文件名”这样的路径保存到一个隐藏域中，目的是为了将其保存到数据库中。

14.4.4　新建图书

处理好图片上传后，新建图书功能就非常简单了，只是一个普通的表单提交功能，最终只会保存一个图片 URL。表单的主要内容请见代码清单 14-17。

代码清单 14-17：codes\14\book\src\main\resources\templates\book\content.html

```
<form id="dataForm" method="post" th:action="@{/book/save}"
    th:object="${book}" th:enctype="'multipart/form-data'">
    <input type="hidden" name="id" th:value="*{id}"/>
    <input type="text" class="form-control" name="name" th:value="*{name}"/>
    <input type="text" class="form-control" name="author" th:value="*{author}"/>
    <input type="text" class="form-control" name="price" th:value="*{price}"/>
    <input type="hidden" value="" id="coverPath" name="cover" th:value="*{cover}"/>
    <img id="coverImg" th:src="*{cover}" height="200px"/>
</form>
```

这里的 HTML 代码是笔者删减后的代码，除去图片上传的功能外，这只是一个非常简单的表单。控制器的 save 方法请见代码清单 14-18。

代码清单 14-18：codes\14\book\src\main\java\org\crazyit\book\controller\BookController.java

```
    @RequestMapping("/save")
    public String save(@ModelAttribute Book book) throws IOException {
        bookService.save(book);
        return "redirect:/book/list";
    }
```

BookSevice 的 save 方法仅仅调用了 BookDao 的 save 方法，这是 JpaRepository 提供的方法，并不需要我们自己定义与实现。

14.4.5 查看与修改图书

我们直接使用新建界面（即 content.html），按照我们的交互方式，实现修改功能。基本上不需要对页面进行修改，修改与查看的控制器方法请见代码清单 14-19。

代码清单 14-19：codes\14\book\src\main\java\org\crazyit\book\controller\BookController.java

```
    /**
     * 打开修改界面
     */
    @GetMapping("/edit/{bookId}")
    public String edit(@PathVariable Integer bookId, @ModelAttribute Book book,
            Model model) {
        model.addAttribute("book", bookService.findById(bookId));
        return "book/content";
    }

    /**
     * 打开查看界面
     */
    @GetMapping("/view/{bookId}")
    public String view(@PathVariable Integer bookId, Model model) {
        model.addAttribute("book", bookService.findById(bookId));
        model.addAttribute("readonly", true);
        return "book/content";
    }
```

实现查看功能时，额外将一个“readonly”变量传到界面，界面的表单域就会根据该变量的值显示为只读的或不可用的，请见代码清单 14-20。

代码清单 14-20：codes\14\book\src\main\resources\templates\book\content.html

```
<form id="dataForm" method="post" th:action="@{/book/save}"
    th:object="${book}" th:enctype="'multipart/form-data'">
    <input type="hidden" name="id" th:value="*{id}"/>
    <input type="text" th:readonly="${readonly}" class="form-control" name="name"
th:value="*{name}"/>
    <input type="text" th:readonly="${readonly}" class="form-control" name=
"author" th:value="*{author}"/>
    <input type="text" th:readonly="${readonly}" class="form-control" name="price"
th:value="*{price}"/>
    <input type="button" class="btn btn-primary form-control" value="上传"
th:disabled="${readonly}" id="uploadButton"/>
    <input id="cover" th:disabled="${readonly}" type="file" class="form-control"/>
    <input type="hidden" value="" id="coverPath" name="cover" th:value=
"*{cover}"/>
    <img id="coverImg" th:src="*{cover}" height="200px"/>
    <input type="submit" class="btn btn-primary" th:hidden="${readonly}" value="
保 存"/>
</form>
```

代码清单 14-20 中的粗体字代码，会判断 readonly 变量的值是否为 true，来决定是否将表单域设置为只读的或不可用的。

14.4.6 删除图书

删除功能较为简单，仅需要将相应的 Book 的 isDelete 设置为 true，然后再保存一次即可。代码清单 14-21 为控制器及 Service 的实现。

代码清单 14-21：codes\14\book\src\main\java\org\crazyit\book\controller\BookController.java

```
@Controller
@RequestMapping("/book")
public class BookController {

    @Autowired
    private BookService bookService;

    @RequestMapping("/delete/{page}")
    public String delete(@RequestParam Integer[] ids, @PathVariable Integer page) {
        bookService.delete(ids);
        return "redirect:/book/list?page=" + page;
    }
}
```

codes\14\book\src\main\java\org\crazyit\book\service\BookService.java

```
@Service
public class BookService {

    @Autowired
    private BookDao bookDao;

    public Book findById(Integer bookId) {
        return bookDao.findById(bookId).get();
    }

    public void delete(Integer[] ids) {
        for(Integer id : ids) {
            Book book = findById(id);
            book.setIsDelete(true);
            save(book);
        }
    }

}
```

我们为列表页面添加一个表单，然后在用户单击“删除”按钮时，提供这个表单。列表页面的内容大致如下：

```
<form id="dataForm" method="post">
    <table class="table" style="border: 1px solid #cccfc9;">
        <tbody th:remove="all-but-first">
            <tr th:each="data : ${datas.content}" th:object="${data}">
                <td scope="row"><input type="checkbox" name="ids" th:value="*{id}"/>
                  </td>
                <td><a href="view.html" th:href="@{/book/view/} + *{id}" target=
                  "_blank" th:text="*{name}">疯狂 Spring Cloud 微服务架构实战</a></td>
                <td th:text="*{author}">杨恩雄</td>
                <td th:text="*{price}">40</td>
                <td><a th:href="@{/book/edit/} + *{id}">修改</a></td>
            </tr>
```

```
        </tbody>
    </table>
</form>
<button type="button" class="btn btn-primary" id="delButton">删除</button></div>
<script>
    $("#delButton").click(function() {
        $("#dataForm").attr("action", "[[ @{/book/delete/} ]][[ ${datas.number} ]]");
        $("#dataForm").submit();
    });
</script>
```

实现逻辑比较简单，只是将表单提交给控制器的 delete 方法，在 delete 方法中会获取要删除的数据 id。在这里要注意一点，为了提升用户体验，在删除时会将当前的页数传到 delete 方法，在删除完数据后，系统会跳转到相应的页中。

至此，图书管理模块的大部分功能都已经实现了，例如数据列表（分页）、新建数据、删除数据。掌握了这些功能的实现方法后，再去实现其他模块的功能，就很简单了，可以直接复制过来使用。

14.5 销售单模块

销售单模块与入库单模块类似，我们在此只讲解如何实现销售单模块。

14.5.1 列表与分页功能

销售单列表中会显示每一张销售单中所销售的全部图书名称，并且会计算销售单的总金额。代码清单 14-22 为 Service 方法的代码。

代码清单 14-22：codes\14\book\src\main\java\org\crazyit\book\service\SaleService.java

```
@Service
public class SaleService {

    @Autowired
    private SaleDao saleDao;

    @Autowired
    private SaleItemDao saleItemDao;

    /**
     * 分页查询数据
     */
    public Page<Sale> findAll(Pageable pageable) {
        Page<Sale> datas = saleDao.findAll(pageable);
        // 设置明细的图书名称
        List<Sale> sales = datas.getContent();
        for(Sale s : sales) {
            List<SaleItem> items = saleItemDao.findBySaleId(s.getId());
            s.initBookNames(items);
            s.initMoney(items);
        }
        return datas;
    }
}
```

在代码清单 14-22 中，findAll 方法会先根据分页参数（Pageable）查询全部的销售单数据，然后对销售单数据进行遍历，再去查询每一张销售单下面的全部明细，之后调用 Sale 对象的 initBookNames 与 initMoney 方法，分别设置销售图书名称与销售总金额。这两个方法请见代码清单 14-23。

代码清单 14-23：codes\14\book\src\main\java\org\crazyit\book\entity\Sale.java

```
    /**
     * 设置图书名称，以逗号隔开
     */
    public void initBookNames(List<SaleItem> items) {
        StringBuffer names = new StringBuffer();
        for(int i = 0; i < items.size(); i++) {
            SaleItem item = items.get(i);
            String name = "《" + item.getBook().getName() + "》";
            if(i == 0) {
                names.append(name);
            } else {
                names.append(", " + name);
            }
        }
        this.bookNames = names.toString();
    }

    /**
     * 计算销售单的总金额
     */
    public void initMoney(List<SaleItem> items) {
        BigDecimal result = new BigDecimal(0);
        for(SaleItem item : items) {
            result = result.add(item.getPrice());
        }
        this.money = result;
    }
```

在控制器方法中，直接按照图书管理的方式，传入分页参数，来设置分页结果，如代码清单 14-24 所示。

代码清单 14-24：codes\14\book\src\main\java\org\crazyit\book\controller\SaleController.java

```
@Controller
@RequestMapping("/sale")
public class SaleController {

    @Autowired
    private SaleService saleService;

    @GetMapping("/list")
    public String list(Model model,
            @PageableDefault(size = BookApp.PAGE_SIZE) Pageable pageable) {
        model.addAttribute("datas", saleService.findAll(pageable));
        return "sale/list";
    }
}
```

在列表页面中，对 tbody 下的 tr 进行遍历，如代码清单 14-25 所示。

代码清单 14-25：codes\14\book\src\main\resources\templates\sale\list.html

```
<tbody th:remove="all-but-first">
    <tr th:each="data : ${datas.content}" th:object="${data}">
        <th scope="row"><input type="checkbox"/></th>
        <td><a href="view.html" target="_blank" th:text="*{bookNames}" th:href=
          "@{/sale/view/} + *{id}">
        疯狂 Spring Cloud 微服务架构实战，疯狂 Spring Cloud 微服务架构实战</a></td>
        <td th:text="*{money}">100</td>
        <td th:text="*{createTime}">2018-01-01 00:00:00</td>
    </tr>
</tbody>
```

由于我们在前面编写了一个通用的分页组件，所以在此可以直接引用分页片断，并传递列表的 URL 作为参数，如下所示：

```
<nav class="col-lg-1" th:replace="~{common/page::page('/sale/list')}">
    ...省略
</nav>
```

启动服务器，在浏览器中访问：http://localhost:8080/sale/list，结果如图 14-12 所示。



图 14-12 销售单列表

14.5.2 选择图书

在新建销售单时，需要选择图书并将它们放到列表中，为此我们做一个通用的图书选择界面，并为新建销售单界面的“选择图书”按钮设置单击动作，请见代码清单 14-26。

代码清单 14-26：codes\14\book\src\main\resources\templates\sale\content.html

```
    <button type="button" class="btn btn-primary" onClick="document.location.href=
'choose.html'"
    th:onclick="|window.open('@{/book/choose}', '_blank', 'width=700,height=500,
resizable=no')|" >
    选择图书</button>
```

“选择图书”按钮被单击后，会打开“/book/choose”，该控制器会查询所有没有被删除的图书，并返回一个列表界面，该界面的主要内容如代码清单 14-27 所示。

代码清单 14-27：codes\14\book\src\main\resources\templates\book\choose.html

```
<tbody th:remove="${datas == null} ? none : all-but-first">
    <tr th:each="data : ${datas.content}" th:object="${data}">
        <th scope="row"><input type="checkbox"
        th:onclick="|selectBook(this, '${data.id}', '${data.name}', '${data.price}')
         |"/></th>
```

```
            <td th:text="*{name}">疯狂 Spring Cloud 微服务架构实战</td>
            <td th:text="*{author}">杨恩雄</td>
            <td th:text="*{price}">40</td>
        </tr>
    </tbody>
    <script>
        function selectBook(checkbox, id, name, price) {
            if(checkbox.checked) {
                window.opener.addBook(id, name, price);
            }
        }
    </script>
```

由于这个图书选择界面，需要提供给销售单模块与入库单模块使用，因此要考虑通用性。当列表中的复选框被选中时，调用父窗口的 addBook 方法，至于 addBook 方法如何实现，则由具体的打开窗口决定，本例中的打开窗口为“新建销售单”，因此还要实现 addBook 方法，请见代码清单 14-28。

代码清单 14-28：codes\14\book\src\main\resources\templates\sale\content.html

```
<tbody id="bookDatas">

</tbody>
<script th:inline="JavaScript">
    /*[+
    // 初始化入库单明细
    var items = [[${items}]];
    if(items == null) {
        items = [];
    }

    // 刷新图书列表
    function refreshBookTable() {
        $("#bookDatas").empty();
        for(var i = 0; i < items.length; i++) {
            var item = items[i];
            var bookId = item.book.id;
            var tr = $("<tr></tr>");
            var td1 = $("<td>" + item.book.name +"</td>");
            // 新建数量的 text
            var amountText = $("<input type='text' class='form-control'/>");
            amountText.val(item.amount);
            // 创建第 2 个 td,
            var td2 = $("<td></td>").append(amountText);
            // 删除按钮
            var delButton = $("<input type='button' class='btn btn-primary' value='
              删除'/>");
            var td3 = $("<td></td>").append(delButton);
            tr.append(td1).append($("<td>" + item.book.price +"</td>")).append
              (td2).append(td3);
            $("#bookDatas").append(tr);
        }
    }

    // 添加一本书到列表
    function addBook(id, name, price) {
```

```
        // 判断图书是否已经存在列表中
        for(var i = 0; i < items.length; i++) {
            var item = items[i];
            if(id == item.book.id) {
                return;
            }
        }
        var newItem = {"book": {"id":"","name":"","price":""}, "amount": 0};
        newItem.book.id = id;
        newItem.book.name = name;
        newItem.book.price = price;
        items.push(newItem);
        refreshBookTable();
    }

    +]*/
</script>
```

下面简单介绍该方法的实现逻辑。在 JavaScript 中定义一个 items 的数组，每添加一本图书，就往这个数组里面 push 一条数据，然后调用 refreshBookTable 方法，刷新图书列表。refreshBookTable 方法会创建多个 HTML 节点，最后形成一个 tr 节点，将其添加到 tbody 中。新建销售单与选择图书界面如图 14-13 所示。



图 14-13 选择图书

下面，我们实现删除选中图书与修改数量功能。

14.5.3 删除已选图书与修改数量

在新建销售单界面，还需要实现删除图书与修改数量两个功能。添加相应的 JavaScript 方法，请见代码清单 14-29。

代码清单 14-29：codes\14\book\src\main\resources\templates\sale\content.html

```
<script th:inline="JavaScript">
    /*[+
    // 初始化入库单明细
    var items = [[${items}]];
    if(items == null) {
```

仅供非商业用途或交流学习使用


```
        items = [];
    }
    refreshBookTable();
    countTotal();

    // 刷新图书列表
    function refreshBookTable() {
        $("#bookDatas").empty();
        var readonly = [[${readonly}]];
        for(var i = 0; i < items.length; i++) {
            var item = items[i];
            var bookId = item.book.id;
            var tr = $("<tr></tr>");
            var td1 = $("<td>" + item.book.name +"</td>");
            // 新建数量的 text
            var amountText = $("<input type='text' class='form-control' onBlur=
              'changeAmount(this, " + bookId + ")'/>");
            amountText.val(item.amount);
            // 创建第 2 个 td,
            var td2 = $("<td></td>").append(amountText);
            // 删除按钮
            var delButton = $("<input type='button' class='btn btn-primary' value='
              删除' onclick='remove(this, " + bookId + ")'/>");
            // 只是查看的话，隐藏删除按钮
            if(readonly) {
                delButton.attr("disabled", true);
                amountText.attr("readonly", true);
            }
            var td3 = $("<td></td>").append(delButton);
            tr.append(td1).append($("<td>" + item.book.price +"</td>")).append
              (td2).append(td3);
            $("#bookDatas").append(tr);
        }
    }

    // 计算总价
    function countTotal() {
        var totalPrice = 0;
        for(var i = 0; i < items.length; i++) {
            var item = items[i];
            var price = item.book.price;
            var amount = item.amount;
            totalPrice = totalPrice + (price * amount);
        }
        $("#totalPrice").val(totalPrice);
    }

    // 当数量输入框失去焦点时，修改数据
    function changeAmount(text, bookId) {
        var amount = $(text).val();
        // 将数据设置到对象中
        for(var i = 0; i < items.length; i++) {
            var item = items[i];
            if(item.book.id == bookId) {
                item.amount = amount;
            }
        }
```

仅供非商业用途或交流学习使用


```
        // 计算总价
        countTotal();
    }

    // 删除一条记录
    function remove(button, bookId) {
        // 删除 tr 元素
        $(button).parent().parent().remove();
        // 删除数据
        for(var i = 0; i < items.length; i++) {
            var item = items[i];
            if(item.book.id == bookId) {
                items.splice(i, 1);
                return;
            }
        }
    }
    +]*/

</script>
```

这里需要关注两个方法：remove 与 changeAmount，remove 方法将当前单击的按钮所在的 tr 删除，然后将该条数据从 items 中移除，而 changeAmount 方法，会在数量输入框失去焦点时被触发，从而重新计算列表中所有图书的总价。

14.5.4 新建销售单

由于我们在新建销售单界面，使用了一个数组来保存选择的图书，那么在提交表单时，将这个数组转换为 JSON 字符串，并提交到控制器就可以完成新建销售单的功能了。提交表单的方法请见代码清单 14-30。

代码清单 14-30：codes\14\book\src\main\resources\templates\sale\content.html

```
<button type="button" class="btn btn-primary" onclick="submitForm()">保存</button>
<script th:inline="JavaScript">
    /*[+
    // 初始化入库单明细
    var items = [[${items}]];
    if(items == null) {
        items = [];
    }
    ... 省略其他方法

    // 提交表单
    function submitForm() {
        var jsonStr = JSON.stringify(items);
        // 将 JSON 设置到隐藏域中提交
        $("#itemJson").val(jsonStr);
        // 提交表单
        $("#dataForm").submit();
    }
    +]*/
</script>
```

控制器方法请见代码清单 14-31。

仅供非商业用途或交流学习使用

代码清单 14-31：codes\14\book\src\main\java\org\crazyit\book\controller\SaleController.java

```
@RequestMapping("/save")
public String save(@RequestParam String itemJson) {
    // 将 JSON 转换为 SaleItem 对象
    List<SaleItem> items = JsonUtil.fromJson(itemJson, SaleItem.class);
    saleService.save(items);
    return "redirect:/sale/list";
}
```

在控制器的 save 方法中，调用 JsonUtil 的转换方法，将 JSON 字符串转换为 SaleItem 集合。JsonUtil 的转换方法如代码清单 14-32 所示。

代码清单 14-32：codes\14\book\src\main\java\org\crazyit\book\util\JsonUtil.java

```
public class JsonUtil {

    /**
     * 将 JSON 转换为集合
     */
    public static List fromJson(String json, Class itemClass) {
        try {
            ObjectMapper mapper = new ObjectMapper();
            JavaType javaType = mapper.getTypeFactory()
                    .constructParametricType(List.class, itemClass);
            return mapper.readValue(json, javaType);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

在转换时，使用了 Jackson 的 API，将 JSON 转换为 List。

控制器的 save 方法，最后会调用 SaleService 的 save 方法保存所有的明细数据。save 方法会创建一条销售单数据，并且设置所有的明细，实现如代码清单 14-33 所示。

代码清单 14-33：codes\14\book\src\main\java\org\crazyit\book\service\SaleService.java

```
/**
 * 创建销售单
 */
public void save(List<SaleItem> items) {
    // 保存销售单
    Sale sale = new Sale();
    sale.setCreateTime(new Date());
    saleDao.save(sale);
    // 保存明细
    for(int i = 0; i < items.size(); i++) {
        SaleItem item = items.get(i);
        // 查询 Book 对象
        Book book = bookDao.findById(item.getBook().getId()).get();
        item.setSale(sale);
        item.setBook(book);
        item.setPrice(book.getPrice());
        saleItemDao.save(item);
    }
}
```

仅供非商业用途或交流学习使用

代码清单 14-33 的逻辑为，先创建一个 Sale 对象，再遍历销售明细，根据所选的图书查询 Book 对象，再将 Book 与 Sale 对象设置到 SaleItem 中，最后保存明细。

14.5.5 操作库存

不管是图书销售还是入库，都需要操作库存。下面我们来编写操作库存的逻辑，请见代码清单 14-34。

代码清单 14-34：codes\14\book\src\main\java\org\crazyit\book\service\BookStockService.java

```
@Service
public class BookStockService {

    @Autowired
    private BookStockDao bookStockDao;

    @Autowired
    private BookDao bookDao;

    /**
     * 为一本书创建库存数据
     */
    public BookStock createStock(Integer bookId) {
        BookStock stock = bookStockDao.findByBookId(bookId);
        Book book = bookDao.findById(bookId).get();
        // 没有才创建，保存一本书只创建一条库存记录
        if(stock == null) {
            stock = new BookStock();
            stock.setBook(book);
            stock.setStockNum(new BigDecimal(0));
            BookStock newStock = bookStockDao.save(stock);
            return newStock;
        }
        return stock;
    }

    /**
     * 增加库存
     */
    public void addStock(Integer bookId, Integer amount) {
        // 更新库存
        BookStock stock = bookStockDao.findByBookId(bookId);
        if(stock == null) {
            stock = createStock(bookId);
        }
        // 新库存 = 旧库存 + 数量
        BigDecimal oldStock = stock.getStockNum();
        stock.setStockNum(oldStock.add(new BigDecimal(amount)));
        bookStockDao.save(stock);
    }

    /**
     * 减少库存
     */
    public void substractStock(Integer bookId, Integer amount) {
        // 更新库存
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        BookStock stock = bookStockDao.findByBookId(bookId);
        if(stock == null) {
            stock = createStock(bookId);
        }
        // 新库存 = 旧库存 - 数量
        BigDecimal oldStock = stock.getStockNum();
        stock.setStockNum(oldStock.subtract(new BigDecimal(amount)));
        bookStockDao.save(stock);
    }
}
```

主要关注 addStock 与 substractStock 方法，顾名思义，一个方法用来增加库存，一个用来减少库存。实现逻辑为，先根据 bookId 查找对应的存库记录，如果查找不到对应的库存记录，则调用 createStock 方法创建一条，然后再增加或者减少库存。以下代码在保存销售单时调用 substractStock 方法：

```
public void save(List<SaleItem> items) {
    // 保存销售单
    Sale sale = new Sale();
    sale.setCreateTime(new Date());
    saleDao.save(sale);
    // 保存明细
    for(int i = 0; i < items.size(); i++) {
        ...省略其他代码
        saleItemDao.save(item);
        // 更新库存
        bookStockService.substractStock(book.getId(), item.getAmount());
    }
}
```

本例并没有判断负库存，在正常的业务系统中，如果库存低于 0，则不允许再进行销售，我们在此并没有实现这个逻辑。入库单模块与销售单模块的功能类似，在此不再赘述。

14.6 首页数据查询与排序

首页中主要包含两个功能，查询库存数据与查询最新添加的图书，下面我们实现这两个功能。

14.6.1 库存查询

首页中的查询库存的 DAO 方法与 Service 方法，请见代码清单 14-35。

代码清单 14-35：codes\14\book\src\main\java\org\crazyit\book\dao\BookStockDao.java

```
public interface BookStockDao extends JpaRepository<BookStock, Integer> {

    // 查询前 10 条库存数据，根据 id 排序
    List<BookStock> findTop10ByOrderByIdDesc();
}
```

codes\14\book\src\main\java\org\crazyit\book\service\BookStockService.java

```
@Service
public class BookStockService {

    @Autowired
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    private BookStockDao bookStockDao;

    /**
     * 查询库存数据
     */
    public List<BookStock> findIndexStocks() {
        List<BookStock> stocks = bookStockDao.findTop10ByOrderByIdDesc();
        return stocks;
    }
}
```

在页面中，直接对查询出来的库存数据进行遍历，请见代码清单 14-36。

代码清单 14-36：codes\14\book\src\main\resources\templates\main.html

```
<div class="col-lg-3" style="padding-right: 10px;">
    <div class="card bg-light mb-3">
        <div class="card-header">
            库存摘要
        </div>
        <ul class="list-group list-group-flush" th:remove="all-but-first">
            <li class="list-group-item" th:each="stock : ${stocks}"
              th:text="${stock.book.name} + '(' + ${stock.stockNum} + ')'"
              th:class="${stock.stockNum} < 100 ? 'list-group-item text-danger' :
                'list-group-item'">
            疯狂 Spring Cloud 微服务架构实战（100 本）</li>
        </ul>
    </div>
</div>
```

在遍历数据时，有一个判断逻辑，如果存库小于 100，则将 class 设置为“text-danger”。查询结果如图 14-14 所示。



图 14-14 首页中的库存信息

14.6.2 图书查询与轮播

首页中的图书查询的 DAO 与 Service 方法，请见代码清单 14-37。

代码清单 14-37：codes\14\book\src\main\java\org\crazyit\book\dao\BookDao.java

```
public interface BookDao extends JpaRepository<Book, Integer> {

    // 查询前 3 条 isDelete 的数据，并且根据 id 排序
    List<Book> findTop3ByIsDeleteOrderByIdDesc(boolean isDelete);
}
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

codes\14\book\src\main\java\org\crazyit\book\service\BookService.java

```
@Service
public class BookService {

    @Autowired
    private BookDao bookDao;

    /**
     * 查询首页书本数据
     */
    public List<Book> findIndexBooks() {
        return bookDao.findTop3ByIsDeleteOrderByIdDesc(false);
    }
}
```

页面逻辑请见代码清单 14-38。

代码清单 14-38：codes\14\book\src\main\resources\templates\main.html

```
<div id="carouselExampleIndicators" class="carousel slide" data-interval="5000">
    <ol class="carousel-indicators" th:remove="all-but-first">
        <li th:each="book : ${books}" th:class="${bookStat.index == 0} ? active"
          th:data-slide-to="${bookStat.index}"
          data-target="#carouselExampleIndicators" data-slide-to="0" class=
            "active"></li>
    </ol>
    <div class="carousel-inner" th:remove="all-but-first"
      style="margin-left: 60px;">
        <div th:each="book : ${books}"
            th:class="${bookStat.index == 0} ? |carousel-item active| : |carousel-item|"
            class="carousel-item active">
            <img class="d-block" width="500" th:src="@{${book.cover}}" src="../
             static/images/slide1.jpg">
        </div>
    </div>
    <a class="carousel-control-prev" href="#carouselExampleIndicators" role="button"
      data-slide="prev">
        <img src="../static/images/back.png" th:src="@{/images/back.png}"/>
    </a>
    <a class="carousel-control-next" href="#carouselExampleIndicators" role="button"
      data-slide="next">
        <img src="../static/images/next.png" th:src="@{/images/next.png}"/>
    </a>
</div>
```

代码清单 14-38 是一个图片轮播组件，注意粗体字部分的代码，在这里对数据进行遍历，并根据具体的循环状态来设置不同的 class。图片轮播效果如图 14-15 所示。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



图 14-15 图片轮播

14.6.3 数据排序

前面讲解过分页的实现，在此，我们以一个简单的列表，介绍一下 Spring Data 提供的排序功能。Service 与控制器的实现请见代码清单 14-39。

代码清单 14-39：codes\14\book\src\main\java\org\crazyit\book\service\BookService.java

```
@Service
public class BookService {

    @Autowired
    private BookDao bookDao;

    public Page<Book> testFindAll(Pageable pageable) {
        return bookDao.findAll(pageable);
    }
}
```

codes\14\book\src\main\java\org\crazyit\book\controller\BookController.java

```
@Controller
@RequestMapping("/book")
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping("/test")
    public String testFindAll(Model model, Pageable pageable) {
        model.addAttribute("datas", bookService.testFindAll(pageable));
        return "book/test";
    }
}
```

在页面中，对数据进行遍历，请见代码清单 14-40。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

代码清单 14-40：codes\14\book\src\main\resources\templates\book\test.html

```
<table class="table" style="border: 1px solid #cccfc9;">
    <thead>
        <tr class="table-secondary">
            <th scope="col" width="10%">选择</th>
            <th scope="col" width="40%">
            <a th:href="@{/book/test?sort=} + ${#strings.equals(datas.sort, 'name:
             DESC') ? 'name,ASC' : 'name,DESC'}">书名</a></th>
            <th scope="col" width="20%">作者</th>
            <th scope="col" width="20%">
            <a th:href="@{/book/test?sort=} + ${#strings.equals(datas.sort, 'price:
             DESC') ? 'price,ASC' : 'price,DESC'}">售价</a></th>
            <th scope="col" width="10%">操作</th>
        </tr>
    </thead>
    <tbody th:remove="all-but-first">
        <tr th:each="data : ${datas.content}" th:object="${data}">
            <td scope="row"><input type="checkbox" name="ids" th:value="*{id}"/> </td>
            <td><a href="view.html" th:href="@{/book/view/} + *{id}" target="_blank"
             th:text="*{name}">疯狂 Spring Cloud 微服务架构实战</a></td>
            <td th:text="*{author}">杨恩雄</td>
            <td th:text="*{price}">40</td>
            <td><a th:href="@{/book/edit/} + *{id}">修改</a></td>
        </tr>
    </tbody>
</table>
```

代码清单 14-40 中的粗体字代码，实现了根据书名、价格进行排序功能，并在其中加入了相应的判断，如果当前是 DESC 排序，则 a 节点的 href 为 ASC 排序，这样做的目的是为了便于在两种排序之间进行切换。

如果想根据多个字段排序，则可以使用&分隔字段，例如需要对 name 进行 DESC 排序，对 price 进行 ASC 排序，则可以请求这样的 URL：/list?sort=name,desc&price,asc。

14.7 本章小结

本章带领大家完成了一个图书进销存系统，该系统中综合运用了本书介绍的大部分技术框架。在开发各个模块时，应重点关注 Thymeleaf 的各项特性，例如列表、逻辑判断、片断的使用等。另外，在开发的过程中，尽量不要破坏原型界面。在开发完成本章的进销存系统后，使用浏览器打开已经加入逻辑的界面，可以看到，它们基本上与原型界面一致。本书偏重于实战，各个框架更深入的知识，因限于篇幅，不能全部介绍，但读者可以以本书的知识为基础，去研究学习这些框架更深入的特性。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

電子工業出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY http://www.phei.com.cn
Broadview WWW.BROADVIEW.COM.CN
博文视点 · IT出版旗舰品牌

十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

以书为证彰显卓越品质

《C++Primer（中文版）（第5版）》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目，被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂，他们的名字熠熠生辉，与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治，成就**博**古通今、**文**圆质方、**视**角独特、**点**石成金之计算机图书的风向标杆：博文视点。

“凤翱翔于千仞兮，非梧不栖”，博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟，与大师并列于IT专业出版之巅。

英雄帖

江湖风云起，代有才人出。
IT界群雄并起，逐鹿中原。
博文视点诚邀天下技术英豪加入，
指点江山，激扬文字
传播信息技术，分享IT心得

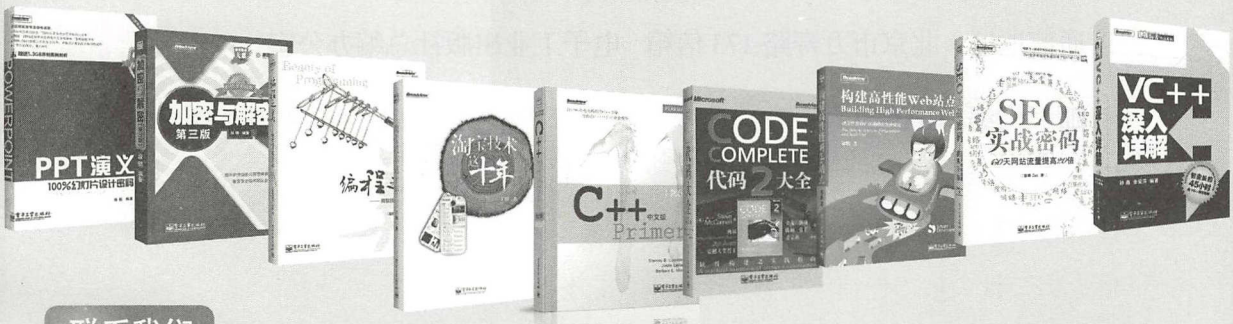
• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版，拥有丰富的与技术图书作者合作的经验，并参照IT技术图书的特点，打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持：

善待作者——我们会把出版流程整理得清晰简明，为作者提供优厚的稿酬服务，解除作者的顾虑，安心写作，展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯，并会参照作者实际的工作、生活节奏，量身制定写作计划，确保合作顺利进行。

提升作者——我们打造精品图书，更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力，为作者的事业开拓带来更多的机会。



联系我们

博文视点官网：http://www.broadview.com.cn　　CSDN官方博客：http://blog.csdn.net/broadview2006/

投稿电话：010-51260888　88254368　　投稿邮箱：jsj@phei.com.cn

新浪微博 weibo.com　@博文视点Broadview　　微信公众账号　博文视点Broadview

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传　　真：（010）88254397

E－mail：dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱　电子工业出版社总编办公室

邮　　编：100036

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

看得懂 学得会 做得出

1. 涵盖多个技术框架

本书期望给读者提供一整套的企业应用开发方案，包括目前流行的Spring Boot、Thymeleaf、Spring Data以及多个JavaScript框架，学习完本书知识后，读者掌握的并不是一两个技术框架，而是一整套当前流行的开发技术。

2. 内容深入

本书以Spring Boot与Thymeleaf为核心，以如何使用Thymeleaf为主线，对Thymeleaf这个目前颇为流行的模板框架进行了系统讲解，既展现了它的技术特性，又辅之以原理分析，务求让读者充分掌握这个模板框架。

3. 视频带领入门

看得懂、学得会、做得出是笔者一贯的风格，对于初次接触这些知识的读者，本书提供了配套视频，助力你更快地掌握知识。

阅读此书有任何技术问题或欲获取本书配套资源，可登录http://www.crazyit.org
亦可访问http://www.broadview.com.cn/34924下载本书配套资源

上架建议：计算机/编程语言

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：张月萍（微信：HM1858）
责任编辑：牛 勇
封面设计：侯士卿

ISBN 978-7-121-34924-9
9 787121 349249 >
定价：69.00元